Docket No.: 43876-128 **PATENT**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of	:	
Craig C. HANSEN, et al.	:	
Serial No.: 09/922,319	:	Group Art Unit: 2671
Filed: August 2, 2001	:	Examiner: M. Monestime

For: SYSTEM WITH WIDE OPERAND ARCHITECTURE AND METHOD

LETTER TO DRAFTSMAN

Hon. Assistant Commissioner for Patents
Washington, DC 20231

Sir:

In response to the Office action mailed on September 23, 2002, having a shortened statutory response period set to expire on December 23, 2002, a three (3) month extension of time up to and including March 23, 2003 being submitted herewith this Amendment, please amend the above identified application as follows:

IN THE DRAWINGS

Please add Figures 12A-34E in accordance with the attached sheets of Figures 12A-20E.

If there are any outstanding issues that might be resolved by an interview or an Examiner's amendment, the Examiner is requested to call Applicants' attorney at the telephone number shown below.

Respectfully submitted,
McDERMOTT, WILL & EMERY

Date: March 24, 2003 By: _____
Lawrence T. Cullen
Registration No. 44,489

600 13th Street, N.W. , Suite 1200
Washington, D.C. 20006-3096
Telephone: (202) 756-8000
Facsimile: (202) 756-8087

1260

Operation codes

W.SWITCH.B	Wide switch big-endian
W.SWITCH.L	Wide switch little-endian

Selection

class	op	order
Wide switch	W.SWITCH	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

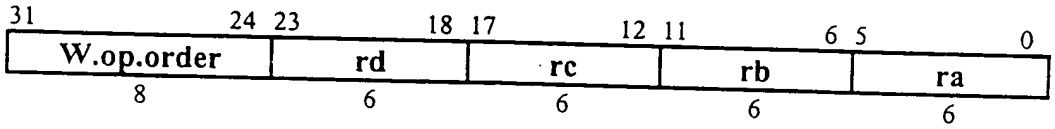


FIG. 12A

1230

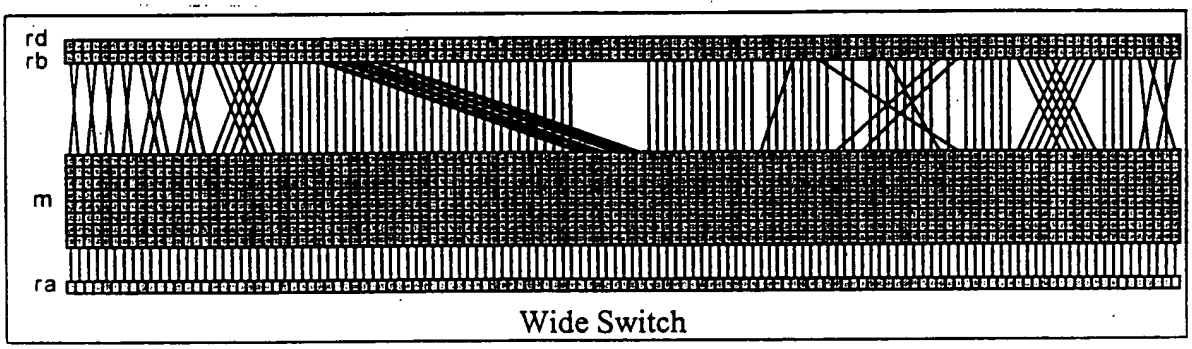


FIG. 12B

1250

Definition

```
def WideSwitch(op,rd,rc,rb,ra)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    if c1..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    elseif c6..0 ≠ 0 then
        VirtAddr ← c and (c-1)
        w ← wsize ← (c and (0-c)) || 0^1
    else
        VirtAddr ← c
        w ← wsize ← 128
    endif
    msize ← 8*wsize
    lwsize ← log(wsize)
    case op of
        W.SWITCH.B:
            order ← B
        W.SWITCH.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    db ← d || b
    for i ← 0 to 127
        j ← 0 || i lwsize-1..0
        k ← m7*w+j||m6*w+j||m5*w+j||m4*w+j||m3*w+j||m2*w+j||mw+j||mj
        l ← i7..lwsize || jlwsize-1..0
        al ← dbl
    endfor
    RegWrite(ra, 128, a)
enddef
```

FIG. 12C

1280

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 12D

1310

Operation codes

W.TRANSLATE.8.B	Wide translate bytes big-endian
W.TRANSLATE.16.B	Wide translate doublets big-endian
W.TRANSLATE.32.B	Wide translate quadlets big-endian
W.TRANSLATE.64.B	Wide translate octlets big-endian
W.TRANSLATE.8.L	Wide translate bytes little-endian
W.TRANSLATE.16.L	Wide translate doublets little-endian
W.TRANSLATE.32.L	Wide translate quadlets little-endian
W.TRANSLATE.64.L	Wide translate octlets little-endian

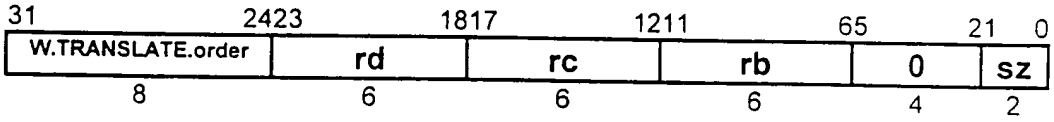
Selection

class	size	order
Wide translate	8 16 32 64	B L

Format

W.TRANSLATE.size.order rd=rc,rb

rd=wtranslatesizeorder(rc,rb)



sz ← log(size) - 3

FIG. 13A

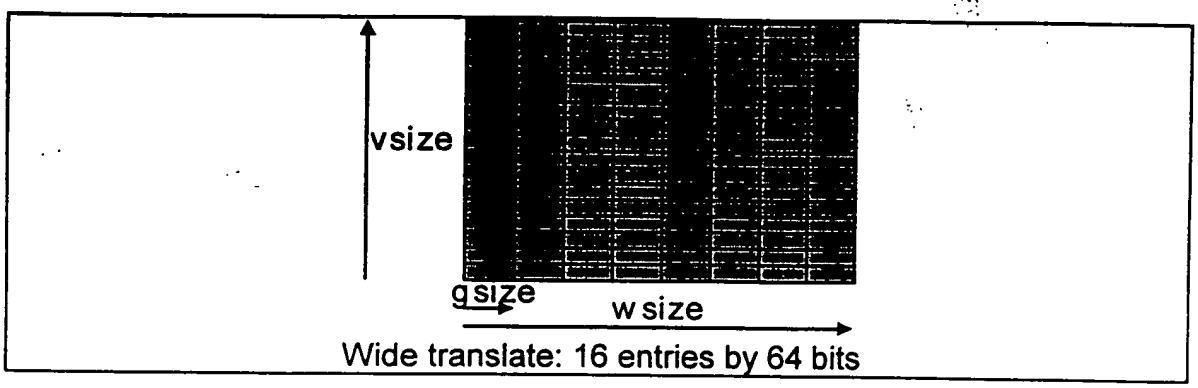


FIG. 13B

1350

Definition

```
def WideTranslate(op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    if c[lgsize-4..0] ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if c[4..lgsize-3] ≠ 0 then
        wsize ← (c and (0-c)) || 0^3
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    lwsize ← log(wsize)
    if t[lwsize+4..lwsize-2] ≠ 0 then
        msize ← (t and (0-t)) || 0^4
        VirtAddr ← t and (t-1)
    else
        msize ← 256*wsize
        VirtAddr ← t
    endif
    case op of
        W.TRANSLATE.B:
            order ← B
        W.TRANSLATE.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    vsize ← msize/wsize
    lvsize ← log(vsize)
    for i ← 0 to 128-gsize by gsize
        j ← ((order=B)^lvsize)^(b[lvsize-1+i..i]))*wsize+i[lwsize-1..0]
        a[gsize-1+i..i] ← m[j+gsize-1..j]
    endfor
    RegWrite(rd, 128, a)
enddef
```

FIG. 13C

1380

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 13D

W.MUL.MAT.8.B	Wide multiply matrix signed byte big-endian
W.MUL.MAT.8.L	Wide multiply matrix signed byte little-endian
W.MUL.MAT.16.B	Wide multiply matrix signed doublet big-endian
W.MUL.MAT.16.L	Wide multiply matrix signed doublet little-endian
W.MUL.MAT.32.B	Wide multiply matrix signed quadlet big-endian
W.MUL.MAT.32.L	Wide multiply matrix signed quadlet little-endian
W.MUL.MAT.C.8.B	Wide multiply matrix signed complex byte big-endian
W.MUL.MAT.C.8.L	Wide multiply matrix signed complex byte little-endian
W.MUL.MAT.C.16.B	Wide multiply matrix signed complex doublet big-endian
W.MUL.MAT.C.16.L	Wide multiply matrix signed complex doublet little-endian
W.MUL.MAT.M.8.B	Wide multiply matrix mixed-signed byte big-endian
W.MUL.MAT.M.8.L	Wide multiply matrix mixed-signed byte little-endian
W.MUL.MAT.M.16.B	Wide multiply matrix mixed-signed doublet big-endian
W.MUL.MAT.M.16.L	Wide multiply matrix mixed-signed doublet little-endian
W.MUL.MAT.M.32.B	Wide multiply matrix mixed-signed quadlet big-endian
W.MUL.MAT.M.32.L	Wide multiply matrix mixed-signed quadlet little-endian
W.MUL.MAT.P.8.B	Wide multiply matrix polynomial byte big-endian
W.MUL.MAT.P.8.L	Wide multiply matrix polynomial byte little-endian
W.MUL.MAT.P.16.B	Wide multiply matrix polynomial doublet big-endian
W.MUL.MAT.P.16.L	Wide multiply matrix polynomial doublet little-endian
W.MUL.MAT.P.32.B	Wide multiply matrix polynomial quadlet big-endian
W.MUL.MAT.P.32.L	Wide multiply matrix polynomial quadlet little-endian
W.MUL.MAT.U.8.B	Wide multiply matrix unsigned byte big-endian
W.MUL.MAT.U.8.L	Wide multiply matrix unsigned byte little-endian
W.MUL.MAT.U.16.B	Wide multiply matrix unsigned doublet big-endian
W.MUL.MAT.U.16.L	Wide multiply matrix unsigned doublet little-endian
W.MUL.MAT.U.32.B	Wide multiply matrix unsigned quadlet big-endian
W.MUL.MAT.U.32.L	Wide multiply matrix unsigned quadlet little-endian

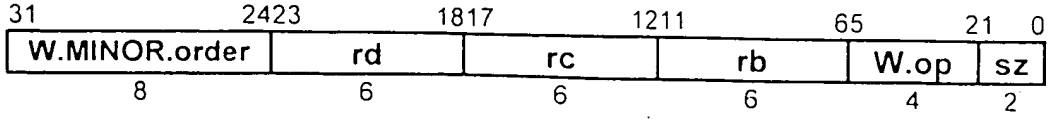
Selection

class	op	type	size	order
multiply	W.MUL.MAT	NONE M U P	8 16 32	B L
		C	8 16	B L

Format

W.op.size.order rd=rc,rb

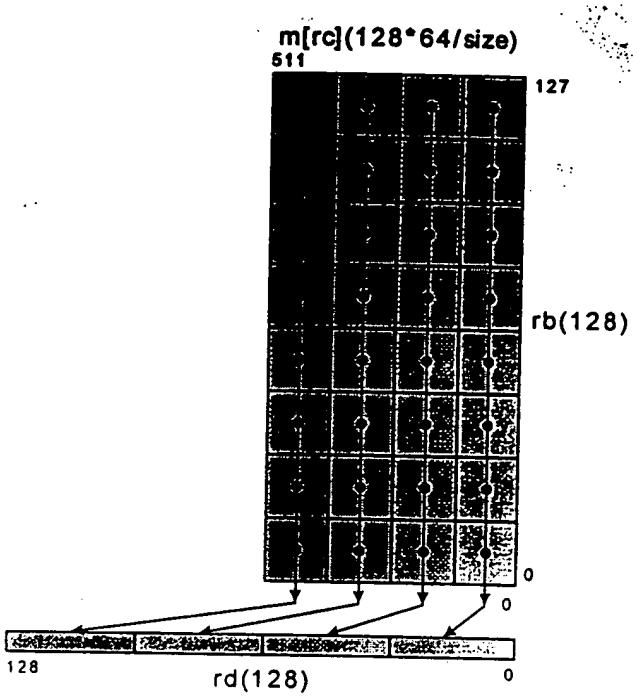
rd=wopsizeorder(rc,rb)



sz ← log(size) - 3

FIG. 14 A

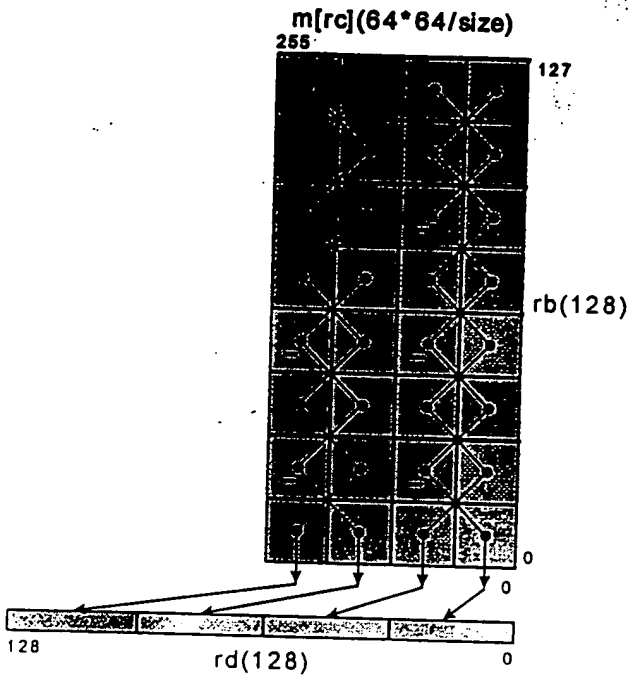
1430



Wide multiply matrix

FIG. 14B

1460



Wide multiply matrix complex

FIG. 14C

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def c ← PolyMultiply(size,a,b) as
    p[0] ← 0^{2*size}
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ a_k ? (0^{size-k} || b || 0^k) : 0^{2*size}
    endfor
    c ← p[size]
enddef

def WideMultiplyMatrix(major,op,gsize,rd,rc,rb)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    if c_{lgsize-4..0} ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    if c_{2..lgsize-3} ≠ 0 then
        wsize ← (c and (0-c)) || 0^4
        t ← c and (c-1)
    else
        wsize ← 64
        t ← a
    endif
    lwsize ← log(wsize)
    if t_{lwsize+6-lgsize..lwsize-3} ≠ 0 then
        msize ← (t and (0-t)) || 0^4
        VirtAddr ← t and (t-1)
    else
        msize ← 128*wsize/gsize
        VirtAddr ← t
    endif
    case major of
        W.MINOR.B:
            order ← B
        W.MINOR.L:
            order ← L
    endcase
    case op of
        W.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64:
            ms ← bs ← 0
        W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32, W.MUL.MAT.M.64:
            ms ← 0
            bs ← 1
        W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32, W.MUL.MAT.64,
        W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32, W.MUL.MAT.C.64:
            ms ← bs ← 1
        W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32, W.MUL.MAT.P.64:
    endcase
```

FIG. 14D

```
    m ← LoadMemory(c,VirtAddr,msize,order)
    h ← 2*gsize

    for i ← 0 to wsize-gsize by gsize
        q[0] ← 0^(2*gsize)
        for j ← 0 to vsize-gsize by gsize
            case op of
                W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32, W.MUL.MAT.P.64:
                    k ← i+wsize*j_{8..lgsize}
                    q[j+gsize] ← q[j] ^ PolyMultiply(gsize,m_{k+gsize-1..k},b_{j+gsize-1..j})
                W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32, W.MUL.MAT.C.64:
                    if (~i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*j_{8..lgsize+1}
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,k,bs,b,j)
                    else
                        k ← i+gsize+wsize*j_{8..lgsize+1}
                        q[j+gsize] ← q[j] - mul(gsize,h,ms,m,k,bs,b,j)
                    endif
                W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32, W.MUL.MAT.64,
                W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32,
W.MUL.MAT.M.64,
                W.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64:
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+wsize*j_{8..lgsize},bs,b,j)
        endfor
        a_{2*gsize-1+2*i..2*i} ← q[vsize]
    endfor
    a_{127..2*wsize} ← 0
    RegWrite(rd, 128, a)
enddef
```

FIG 14D (CONTINUED)

1490

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 14E

1510

Operation codes

W.MUL.MAT.X.B	Wide multiply matrix extract big-endian
W.MUL.MAT.X.L	Wide multiply matrix extract little-endian

Selection

class	op	order
Multiply matrix extract	W.MUL.MAT.X	B L

Format

W.op.order ra=rc,rd,rb

ra=wop(rc,rd,rb)

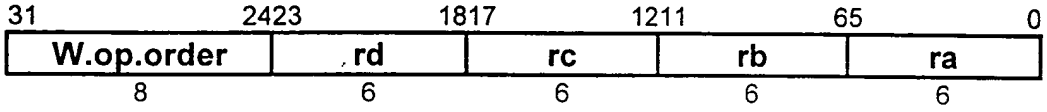


FIG. 15A

1520

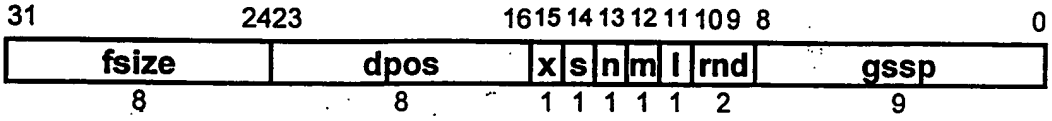
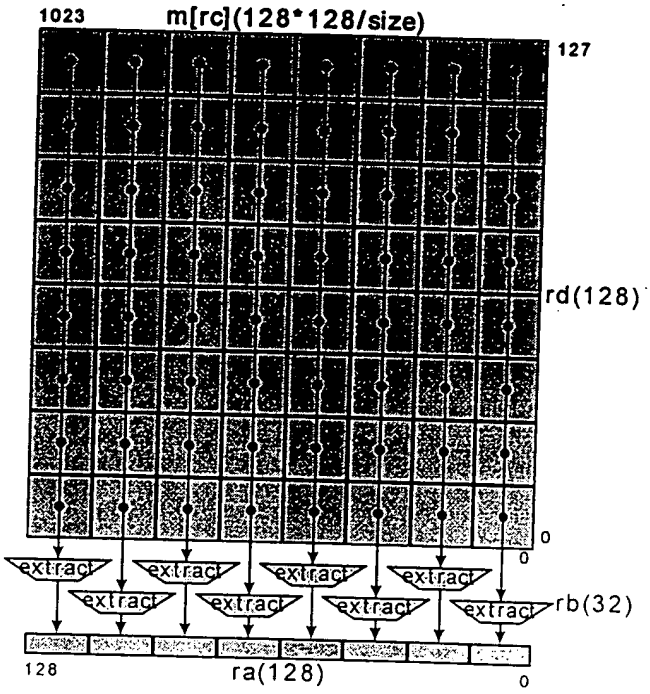

31
2423
16 15 14 13 12 11 10 9 8
0
fsize
dpos
x
s
n
m
l
rnd
gssp
8
8
1 1 1 1 1 2
9

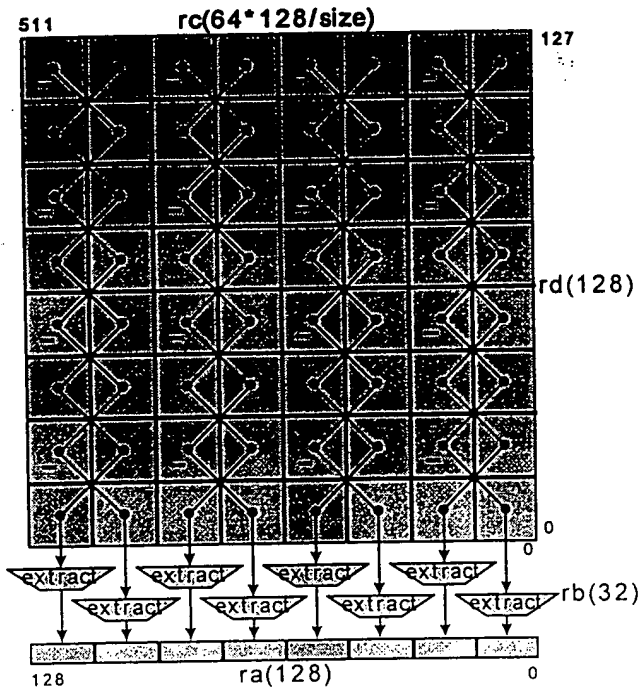
FIG. 15B

1530



Wide multiply matrix extract doublets

FIG. 15C



Wide multiply matrix extract complex doublets

FIG. 15D

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def WideMultiplyMatrixExtract(op,ra,rb,rc,rd)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    case b_{8..0} of
        0..255:
            sgsize ← 128
        256..383:
            sgsize ← 64
        384..447:
            sgsize ← 32
        448..479:
            sgsize ← 16
        480..495:
            sgsize ← 8
        496..503:
            sgsize ← 4
        504..507:
            sgsize ← 2
        508..511:
            sgsize ← 1
    endcase
    l ← b_{11}
    m ← b_{12}
    n ← b_{13}
    signed ← b_{14}
    if c_{3..0} ≠ 0 then
        wsize ← (c and (0-c)) || 0^4
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    if sgsize < 8 then
        gsize ← 8
    elseif sgsize > wsize/2 then
        gsize ← wsize/2
    else
        gsize ← sgsize
    endif
    lgsize ← log(gsize)
    lwsize ← log(wsize)
    if t_{lwsize+6-n-lgsize..lwsize-3} ≠ 0 then
        msize ← (t and (0-t)) || 0^4
        VirtAddr ← t and (t-1)
    else
        msize ← 64*(2-n)*wsize/gsize
        VirtAddr ← t
    endif
```

FIG. 15E

```
vsize ← (1+n)*msize*gsize/wsize
mm ← LoadMemory(c,VirtAddr,msize,order)
h ← (2*gsize) + 7 - lgsize
lmsize ← log(msize)
if (VirtAddr_lmsize-4..0 ≠ 0 then
      raise AccessDisallowedByVirtualAddress
endif
case op of
      W.MUL.MAT.X.B:
            order ← B
      W.MUL.MAT.X.L:
            order ← L
endcase
ms ← signed
ds ← signed ^ m
as ← signed or m
spos ← (b8..0) and (2*gsize-1)
dpos ← (0 || b23..16) and (gsize-1)
r ← spos
sfsize ← (0 || b31..24) and (gsize-1)
tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
fsize ← (tfsize + spos > h) ? h - spos : tfsize
if (b10..9 = Z) & ~signed then
      rnd ← F
else
      rnd ← b10..9
endif
```

FIG. 15E (CONTINUED)

```
for i ← 0 to wsize-gsize by gsize
    q[0] ← 0^(2*gsize+7-lgsize)
    for j ← 0 to vsize-gsize by gsize
        if n then
            if (~i) & j & gsize = 0 then
                k ← i-(j&gsize)+wsize*j8..lgsize+1
                q[j+gsize] ← q[j] + mul(gsize,h,ms,mm,k,ds,d,j)
            else
                k ← i+gsize+wsize*j8..lgsize+1
                q[j+gsize] ← q[j] - mul(gsize,h,ms,mm,k,ds,d,j)
            endif
        else
            q[j+gsize] ← q[j] + mul(gsize,h,ms,mm,i+j*wsize/gsize,ds,d,j)
        endif
    endfor
    p ← q[128]
    case rnd of
        none, N:
            s ← 0^(h-r) || ~p_r || p_r^(r-1)
        Z:
            s ← 0^(h-r) || p_(h-1)^r
        F:
            s ← 0^h
        C:
            s ← 0^(h-r) || 1^r
    endcase
    v ← ((ds & p_(h-1))||p) + (0||s)

        if (v_(h..r+fsize) = (as & v_(r+fsize-1))^(h+1-r-fsize)) or not l then
            w ← (as & v_(r+fsize-1))^(gsize-fsize-dpos) || v_(fsize-1+r..r) || 0^dpos
        else
            w ← (s ? (v_h || ~v_h^(gsize-dpos-1)) : 1^(gsize-dpos)) || 0^dpos
        endif
        a_(size-1+i..i) ← w
    endfor
    a_(127..wsize) ← 0
    RegWrite(ra, 128, a)
enddef
```

FIG ISE (CONTINUED)

1580

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 15F

Operation codes

W.MUL.MAT.X.I.8.B	Wide multiply matrix extract immediate signed bytes big-endian
W.MUL.MAT.X.I.8.L	Wide multiply matrix extract immediate signed bytes little-endian
W.MUL.MAT.X.I.16.B	Wide multiply matrix extract immediate signed doublets big-endian
W.MUL.MAT.X.I.16.L	Wide multiply matrix extract immediate signed doublets little-endian
W.MUL.MAT.X.I.32.B	Wide multiply matrix extract immediate signed quadlets big-endian
W.MUL.MAT.X.I.32.L	Wide multiply matrix extract immediate signed quadlets little-endian
W.MUL.MAT.X.I.64.B	Wide multiply matrix extract immediate signed octlets big-endian
W.MUL.MAT.X.I.64.L	Wide multiply matrix extract immediate signed octlets little-endian
W.MUL.MAT.X.I.C.8.B	Wide multiply matrix extract immediate complex bytes big-endian
W.MUL.MAT.X.I.C.8.L	Wide multiply matrix extract immediate complex bytes little-endian
W.MUL.MAT.X.I.C.16.B	Wide multiply matrix extract immediate complex doublets big-endian
W.MUL.MAT.X.I.C.16.L	Wide multiply matrix extract immediate complex doublets little-endian
W.MUL.MAT.X.I.C.32.B	Wide multiply matrix extract immediate complex quadlets big-endian
W.MUL.MAT.X.I.C.32.L	Wide multiply matrix extract immediate complex quadlets little-endian

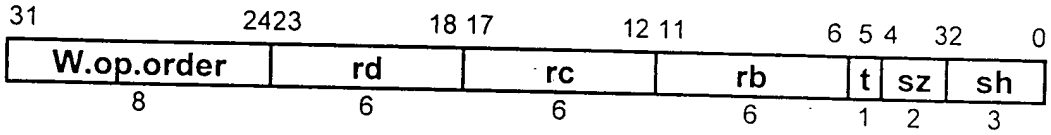
Selection

class	op	type	size	order
wide multiply matrix extract immediate	W.MUL.MAT.X.I	NONE	8 16 32 64	L B
		C	8 16 32	L B

Format

W.op.tsize.order rd=rc,rb,i

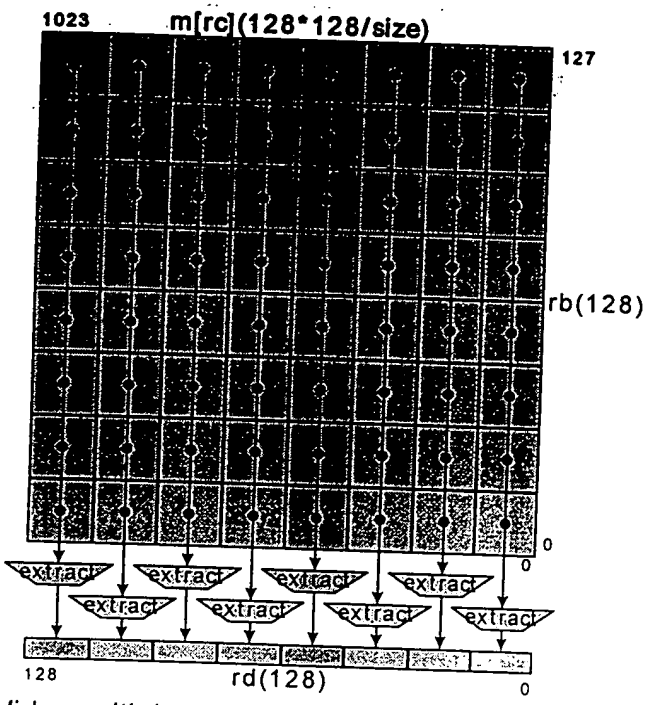
rd=woptsizeorder(rc,rb,i)



sz ← log(size) - 3
assert size+3 ≥ i ≥ size-4
sh ← i - size

FIG. 16A

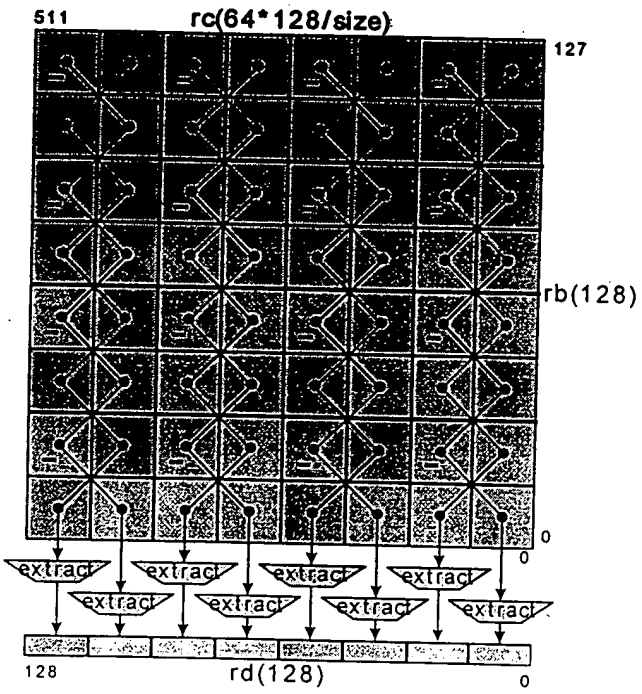
1630



Wide multiply matrix extract immediate doublets

FIG. 16B

1660



Wide multiply matrix extract immediate complex doublets

FIG. 16C

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_size-1+i)^h-size || v_size-1+i..i) * ((ws&w_size-1+j)^h-size || w_size-1+j..j)
enddef

def WideMultiplyMatrixExtractImmediate(op,type,gsize,rd,rc,rb,sh)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    case type of
        NONE:
            if c_lgsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c_3..lgsize-3 ≠ 0 then
                wsize ← (c and (0-c)) || 0^4
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if t_lwsize+6-lgsize..lwsize-3 ≠ 0 then
                msize ← (t and (0-t)) || 0^4
                VirtAddr ← t and (t-1)
            else
                msize ← 128*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← msize*gsize/wsize
        C:
            if c_lgsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c_3..lgsize-3 ≠ 0 then
                wsize ← (c and (0-c)) || 0^4
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if t_lwsize+5-lgsize..lwsize-3 ≠ 0 then
                msize ← (t and (0-t)) || 0^4
                VirtAddr ← t and (t-1)
            else
                msize ← 64*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← 2*msize*gsize/wsize
    endcase
```

1680

```
    case op of
        W.MUL.MAT.X.I.B:
            order ← B
        W.MUL.MAT.X.I.L:
            order ← L
    endcase
    as ← ms ← bs ← 1
    m ← LoadMemory(c,VirtAddr,msize,order)
    h ← (2*gsize) + 7 - lgsize - (ms and bs)
    r ← gsize + (sh[5..2] || sh)
    for i ← 0 to wsize-gsize by gsize
        q[0] ← 0^(2*gsize+7-lgsize)
        for j ← 0 to vsize-gsize by gsize
            case type of
                NONE:
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+wsize*j[8..lgsize],bs,b,j)
                C:
                    if (~i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*j[8..lgsize+1]
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,k,bs,b,j)
                    else
                        k ← i+gsize+wsize*j[8..lgsize+1]
                        q[j+gsize] ← q[j] - mul(gsize,h,ms,m,k,bs,b,j)
                    endif
            endcase
        endfor
        p ← q[vsize]
        s ← 0^(h-r) || ~p_r || p_r^(r-1)
        v ← ((as & p[h-1])||p) + (0||s)
        if (v[h..r+gsize] = (as & v[r+gsize-1])^(h+1-r-gsize) then
            a[gsize-1+i..i] ← v[gsize-1+r..r]
        else
            a[gsize-1+i..i] ← as ? (v_h || ~v_h^(gsize-1)) : 1^gsize
        endif
    endfor
    a[127..wsize] ← 0
    RegWrite(rd, 128, a)
enddef
```

FIG. 16D (CONTINUED)

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG 16E

1710

Operation codes

W.MUL.MAT.C.F.16.B	Wide multiply matrix complex floating-point half big-endian
W.MUL.MAT.C.F.16.L	Wide multiply matrix complex floating-point half little-endian
W.MUL.MAT.C.F.32.B	Wide multiply matrix complex floating-point single big-endian
W.MUL.MAT.C.F.32.L	Wide multiply matrix complex floating-point single little-endian
W.MUL.MAT.F.16.B	Wide multiply matrix floating-point half big-endian
W.MUL.MAT.F.16.L	Wide multiply matrix floating-point half little-endian
W.MUL.MAT.F.32.B	Wide multiply matrix floating-point single big-endian
W.MUL.MAT.F.32.L	Wide multiply matrix floating-point single little-endian
W.MUL.MAT.F.64.B	Wide multiply matrix floating-point double big-endian
W.MUL.MAT.F.64.L	Wide multiply matrix floating-point double little-endian

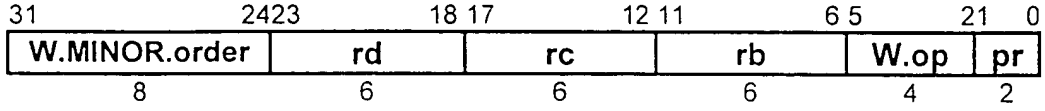
Selection

class	op	type	prec	order
wide multiply matrix	W.MUL.MAT	F	16 32 64	L B
		C.F	16 32	L B

Format

W.op.prec.order rd=rc,rb

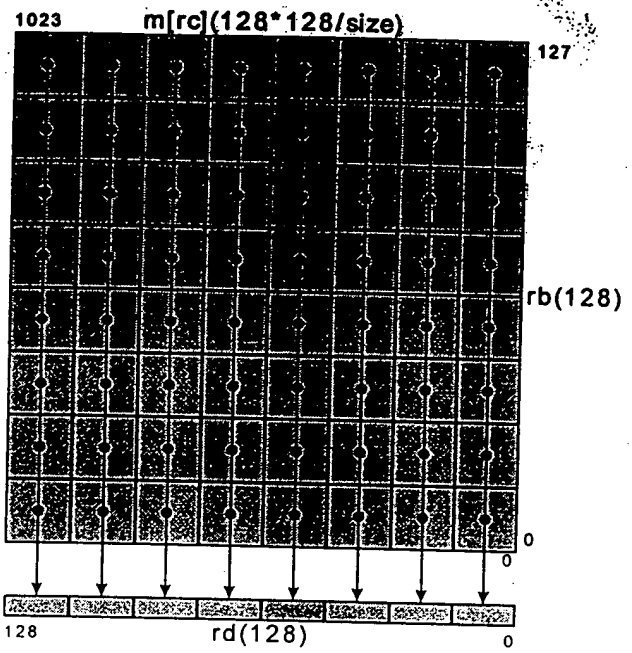
rd=wopprecorder(rc,rb)



pr ← log(prec) - 3

FIG. 17A

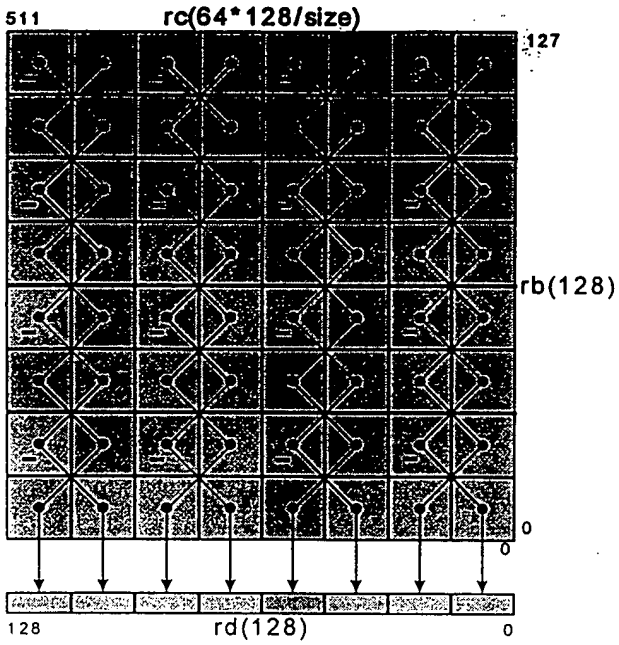
1730



Wide multiply matrix floating-point half

FIG. 17B

1760



Wide multiply matrix complex floating-point half

FIG. 17C

Definition

```
def mul(size,v,i,w,j) as
    mul ← fmul(F(size,v_{size-1+i..i}),F(size,w_{size-1+j..j}))
enddef

def WideMultiplyMatrixFloatingPoint(major,op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lgsize ← log(gsize)
    switch op of
        W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
            if c_{lgsize-4..0} ≠ 0 then
                raise AccessDisallowedByVirtualAddress,
            endif
            if c_{3..lgsize-3} ≠ 0 then
                wsize ← (c and (0-c)) || 0^4
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if t_{lwsize+6-lgsize..lwsize-3} ≠ 0 then
                msize ← (t and (0-t)) || 0^4
                VirtAddr ← t and (t-1)
            else
                msize ← 128*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← msize*gsize/wsize
        W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, W.MUL.MAT.C.F.64:
            if c_{lgsize-4..0} ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c_{3..lgsize-3} ≠ 0 then
                wsize ← (c and (0-c)) || 0^4
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsize ← log(wsize)
            if t_{lwsize+5-lgsize..lwsize-3} ≠ 0 then
                msize ← (t and (0-t)) || 0^4
                VirtAddr ← t and (t-1)
            else
                msize ← 64*wsize/gsize
                VirtAddr ← t
            endif
            vsize ← 2*msize*gsize/wsize
    endcase
```

FIG. 17D

```
    case major of
        M.MINOR.B:
            order ← B
        M.MINOR.L:
            order ← L
    endcase
    m ← LoadMemory(c,VirtAddr,msize,order)
    for i ← 0 to wsize-gsize by gsize
        q[0].t ← NULL
        for j ← 0 to vsize-gsize by gsize
            case op of
                W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
                    q[j+gsize] ← fadd(q[j], mul(gsize,m,i+wsize*j8..lgsize,b,j))
                W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, M.MUL.MAT.C.F.64:
                    if (~i) & j & gsize = 0 then
                        k ← i-(j&gsize)+wsize*j8..lgsize+1
                        q[j+gsize] ← fadd(q[j], mul(gsize,m,k,b,j))
                    else
                        k ← i+gsize+wsize*j8..lgsize+1
                        q[j+gsize] ← fsub(q[j], mul(gsize,m,k,b,j))
                    endif
            endcase
        endfor
        agsize-1+i..i ← q[vsize]
    endfor
    a127..wsize ← 0
    RegWrite(rd, 128, a)
enddef
```

FIG. 17D (CONTINUED)

1790

Exceptions

Floating-point arithmetic
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG 17E

1810

Operation codes

W.MUL.MAT.G.8.B	Wide multiply matrix Galois bytes big-endian
W.MUL.MAT.G.8.L	Wide multiply matrix Galois bytes little-endian

Selection

class	op	size	order
Multiply matrix Galois	W.MUL.MAT.G	8	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

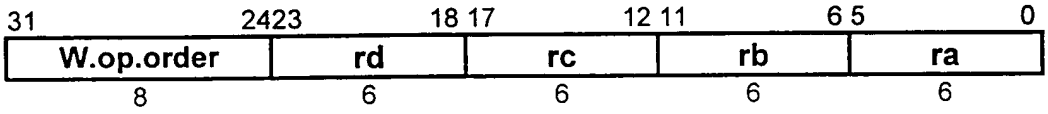
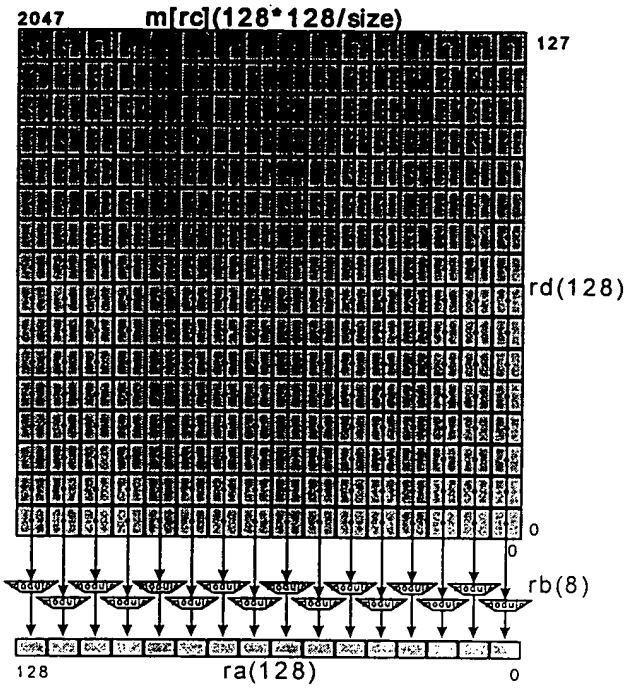


FIG. 18A



Wide multiply matrix Galois byte

FIG. 18B

Definition

```
def c ← PolyMultiply(size,a,b) as
        p[0] ← 0^(2*size)
        for k ← 0 to size-1
                p[k+1] ← p[k] ^ a_k ? (0^(size-k) || b || 0^k) : 0^(2*size)
        endfor
        c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
        p[0] ← a
        for k ← size-1 to 0 by -1
                p[k+1] ← p[k] ^ p[0]_(size+k) ? (0^(size-k) || 1^1 || b || 0^k) : 0^(2*size)
        endfor
        c ← p[size]_(size-1..0)
enddef

def WideMultiplyMatrixGalois(op,gsize,rd,rc,rb,ra)
        d ← RegRead(rd, 128)
        c ← RegRead(rc, 64)
        b ← RegRead(rb, 128)
        lgsize ← log(gsize)
        if c_(lgsize-4..0) ≠ 0 then
                raise AccessDisallowedByVirtualAddress
        endif
        if c_(3..lgsize-3) ≠ 0 then
                wsize ← (c and (0-c)) || 0^4
                t ← c and (c-1)
        else
                wsize ← 128
                t ← c
        endif
        lwsize ← log(wsize)
        if t_(lwsize+6-lgsize..lwsize-3) ≠ 0 then
                msize ← (t and (0-t)) || 0^4
                VirtAddr ← t and (t-1)
        else
                msize ← 128*wsize/gsize
                VirtAddr ← t
        endif
        case op of
                W.MUL.MAT.G.8.B:
                        order ← B
                W.MUL.MAT.G.8.L:
                        order ← L
        endcase
```

FIG. 18C

1860

```
        m ← LoadMemory(c,VirtAddr,msize,order)
        for i ← 0 to wsize-gsize by gsize
            q[0] ← 0^(2*gsize)
            for j ← 0 to vsize-gsize by gsize
                k ← i+wsize*j_{8..lgsize}
                q[j+gsize] ← q[j] ^ PolyMultiply(gsize,m_{k+gsize-1..k},d_{j+gsize-1..j})
            endfor

            a_{gsize-1+i..i} ← PolyResidue(gsize,q[vsize],b_{gsize-1..0})
        endfor
        a_{127..wsize} ← 0
        RegWrite(ra, 128, a)
    enddef
```

FIG. 18C (CONTINUED)

1890

Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 18D

1910

Operation codes

E.MUL.ADD.X	Ensemble multiply add extract
E.CON.X	Ensemble convolve extract

Format

E.op rd@rc,rb,ra

rd=gop(rd,rc,rb,ra)

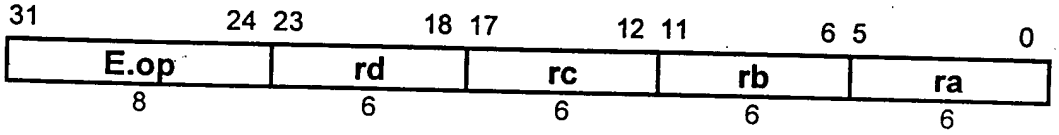


FIG. 19A

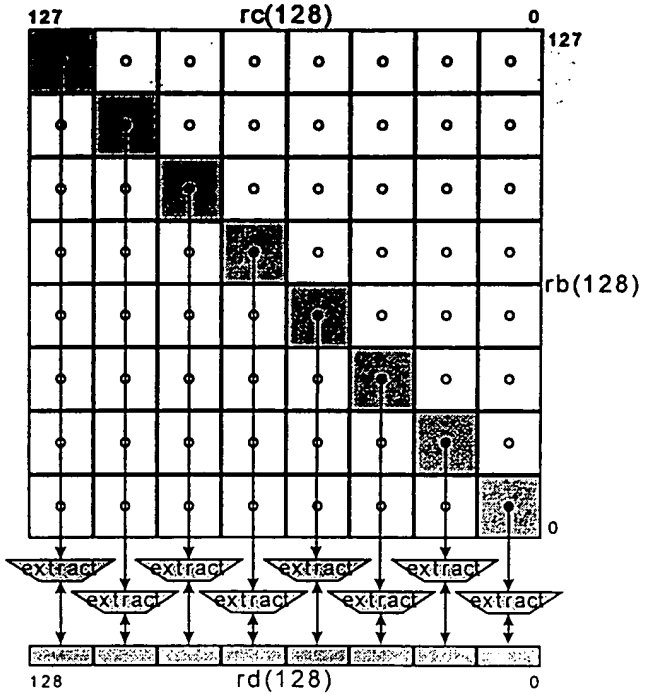
191.0

Figures 19B and 20B has blank fields: should be .

fsize	dpos	x	s	n	m	l	rnd	gssp

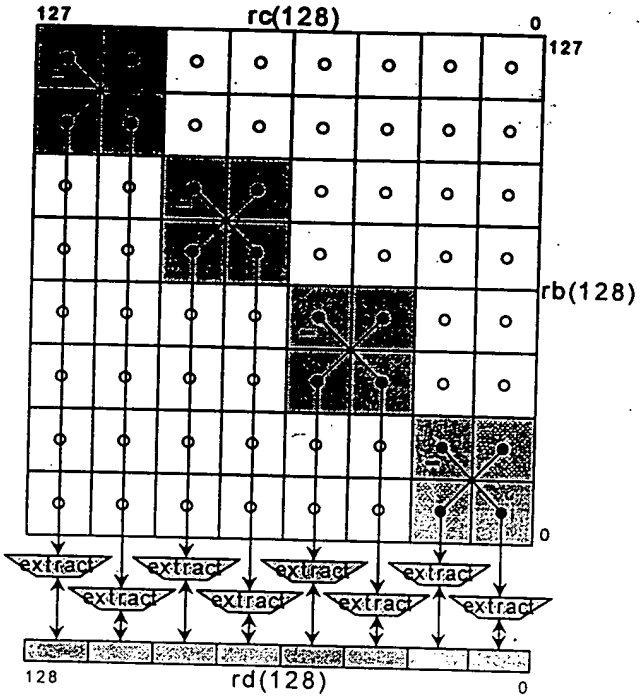
FIG. 19B

1930



Ensemble multiply add extract doublets

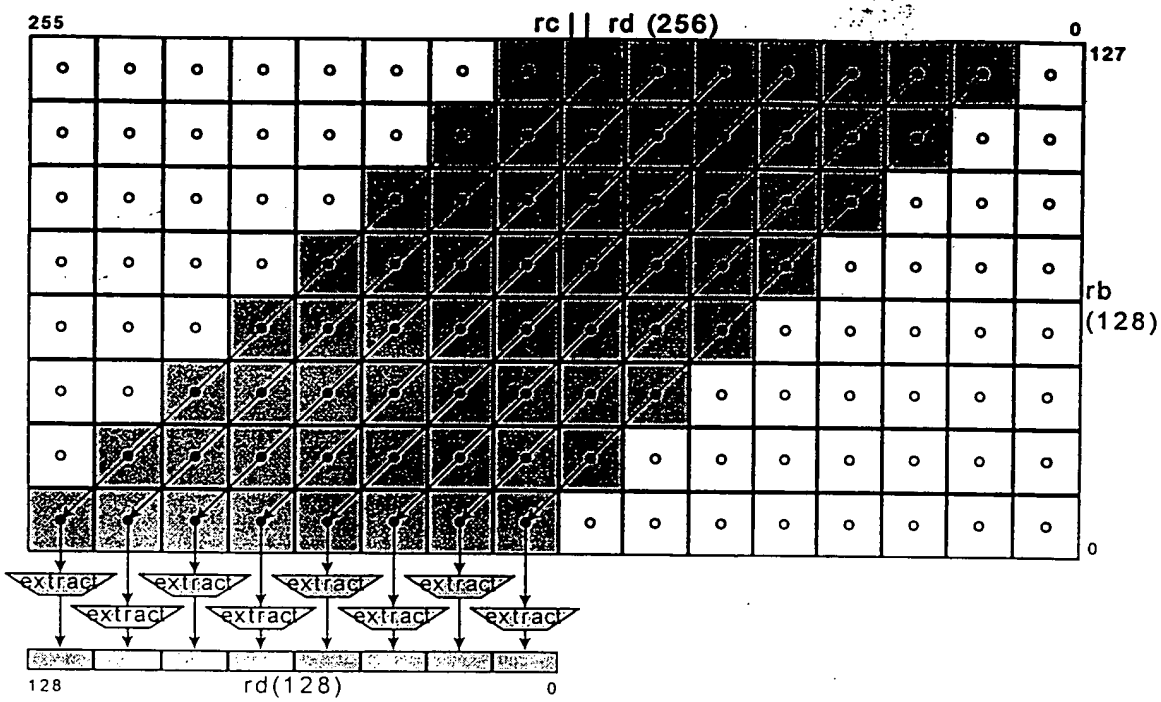
FIG. 19C



Ensemble complex multiply add extract doublets

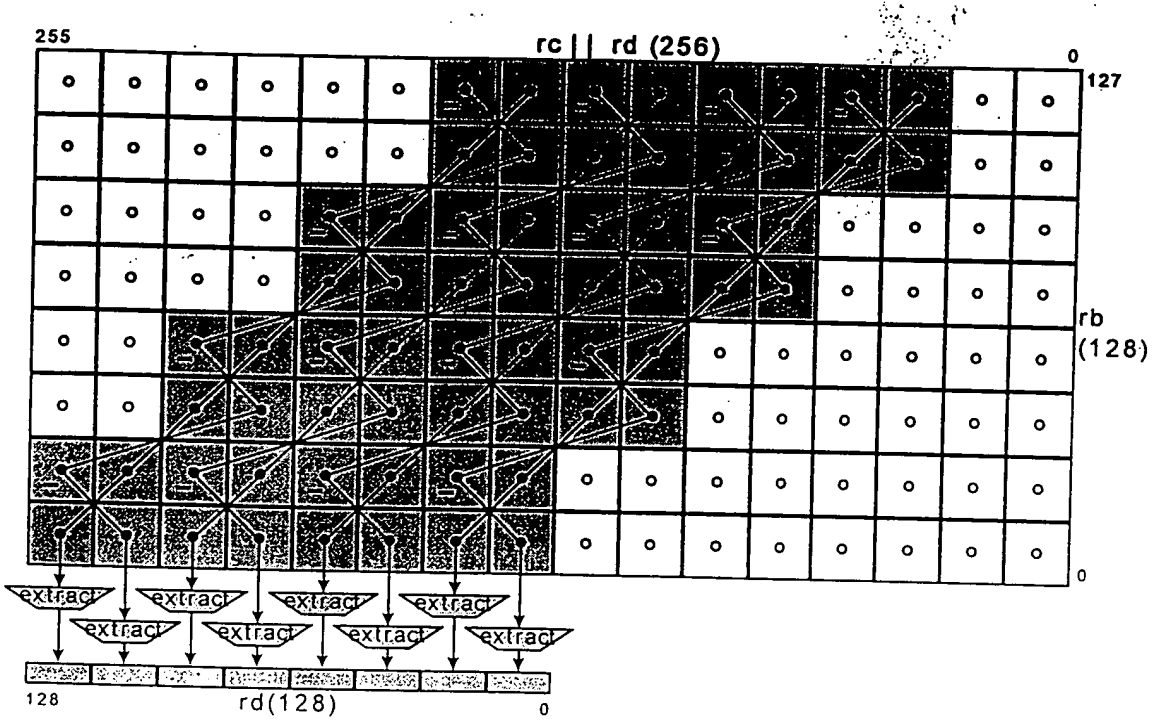
The ensemble-multiply-add-extract instructions (E.MUL.ADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 19D



Ensemble convolve extract doublets

FIG. 19E



Ensemble convolve extract complex doublets

FIG. 19F

1890

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_{size-1+i})^{h-size} || v_{size-1+i..i}) * ((ws&w_{size-1+j})^{h-size} || w_{size-1+j..j})
enddef

def EnsembleExtractInplace(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case a_{8..0} of
        0..255:
            sgsize ← 128
        256..383:
            sgsize ← 64
        384..447:
            sgsize ← 32
        448..479:
            sgsize ← 16
        480..495:
            sgsize ← 8
        496..503:
            sgsize ← 4
        504..507:
            sgsize ← 2
        508..511:
            sgsize ← 1
    endcase
    l ← a_{11}
    m ← a_{12}
    n ← a_{13}
    signed ← a_{14}
    x ← a_{15}
    case op of
        E.CON.X:
            if (sgsize < 8) then
                gsize ← 8
            elseif (sgsize*(n+1)*(x+1) > 128) then
                gsize ← 128/(n+1)/(x+1)
            else
                gsize ← sgsize
            endif
            lgsize ← log(gsize)
            wsize ← 128/(x+1)
            vsize ← 128
            ds ← cs ← signed
            bs ← signed ^ m
            zs ← signed or m or n
            zsize ← gsize*(x+1)
            h ← (2*gsize) + log(vsize) - lgsize
            spos ← (a_{8..0}) and (2*gsize-1)
```

```
        E.MUL.ADD.X:
                if (sgsize < 8) then
                        gsize ← 8
                elseif (sgsize*(n+1)*(x+1) > 128) then
                        gsize ← 128/(n+1)/(x+1)
                else
                        gsize ← sgsize
                endif
                ds ← signed
                cs ← signed ^ m
                zs ← signed or m or n
                zsize ← gsize*(x+1)
                h ← (2*gsize) + n
                spos ← (a8..0) and (2*gsize-1)
endcase
dpos ← (0 || a23..16) and (zsize-1)
r ← spos
sfsize ← (0 || a31..24) and (zsize-1)
tfsize ← (sfsize = 0) or ((sfsize+dpos) > zsize) ? zsize-dpos : sfsize
fsize ← (tfsize + spos > h) ? h - spos : tfsize
if (b10..9 = Z) and not as then
        rnd ← F
else
        rnd ← b10..9
endif
```

FIG. 19G (CONTINUED)

```
for k ← 0 to wsize-zsiz[illegible]hy zsize
    i ← k*gsize/zsi[illegible]
    case op of
        E.CON.X:
            q[0] ← 0^(2*gsize+7-lgsize)
            for j ← 0 to vsize-gsize by gsize
                if n then
                    if (~i) & j & gsize = 0 then
                        q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+128-j,bs,b,j)
                    else
                        q[j+gsize] ← q[j] - mul(gsize,h,ms,m,i+128-j+2*gsize,bs,b,j)
                    endif
                else
                    q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+128-j,bs,b,j)
                endif
            endfor
            p ← q[vsize]
        E.MUL.ADD.X:
            di ← ((ds and d_(k+zsize-1))^(h-zsize-r)||(d_(k+zsize-1..k))||0^r)
            if n then
                if (i and gsize) = 0 then
                    p←mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gsize,cs,c,i+gsize)+di
                else

    p←mul(gsize,h,ds,d,i,cs,c,i+gsize)+mul(gsize,h,ds,d,i,cs,c,i+gsize)+di
                endif
            else
                p ← mul(gsize,h,ds,d,i,cs,c,i) + di
            endif
    endcase
    case rnd of
        N:
            s ← 0^(h-r) || ~p_r || p_r^(r-1)
        Z:
            s ← 0^(h-r) || p_(h-1)^r
        F:
            s ← 0^h
        C:
            s ← 0^(h-r) || 1^r
    endcase
    v ← ((zs & p_(h-1))||p) + (0||s)
    if (v_(h..r+fsize) = (zs & v_(r+fsize-1))^(h+1-r-fsize)) or not (I and (op = E.EXTRACT)) then
        w ← (zs & v_(r+fsize-1))^(zsize-fsize-dpos) || v_(fsize-1+r..r) || 0^dpos
    else
        w ← (zs ? (v_h || ~v_h^(zsize-dpos-1)) : 1^(zsize-dpos)) || 0^dpos
    endif
    z_(zsize-1+k..k) ← w
  endfor
  RegWrite(rd, 128, z)
enddef
```

FIG. 19G (CONTINUED)

2010

Operation codes

E.MUL.X	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCAL.ADD.X	Ensemble scale add extract

Format

E.op ra=rd,rc,rb

ra=eop(rd,rc,rb)

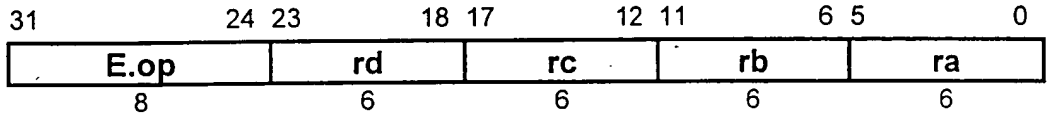


FIG. 20A

2015

Figures 19B and 20B has blank fields: should be .

fsize	dpos	x	s	n	m	l	rnd	gssp

Fig. 20C

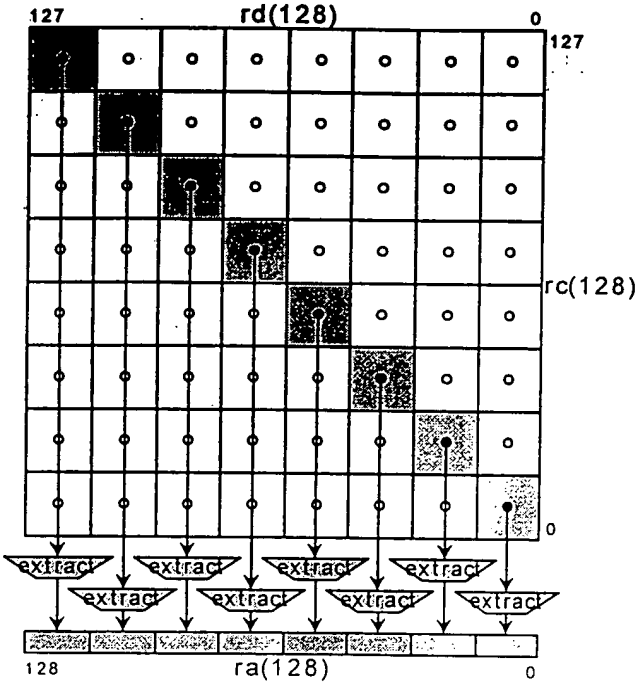
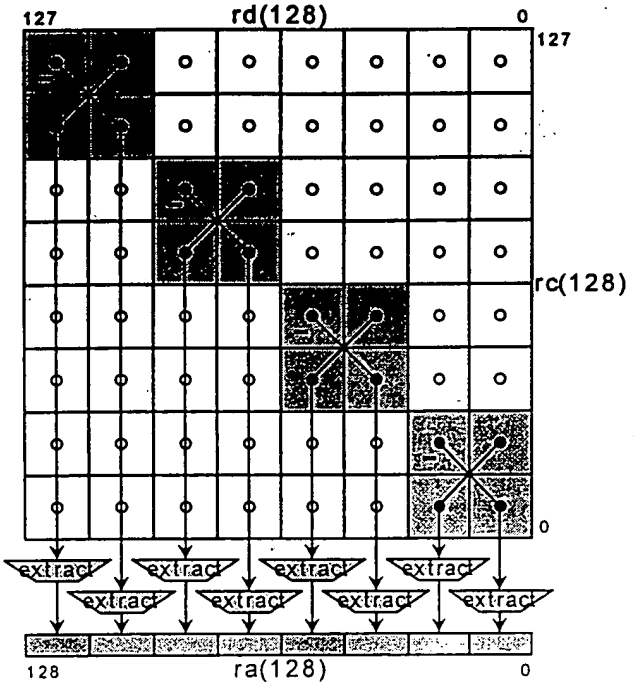

127
rd(128)
0
127
rc(128)
0
extract
extract
extract
extract
extract
extract
extract
extract
128
ra(128)
0

FIG. 20C

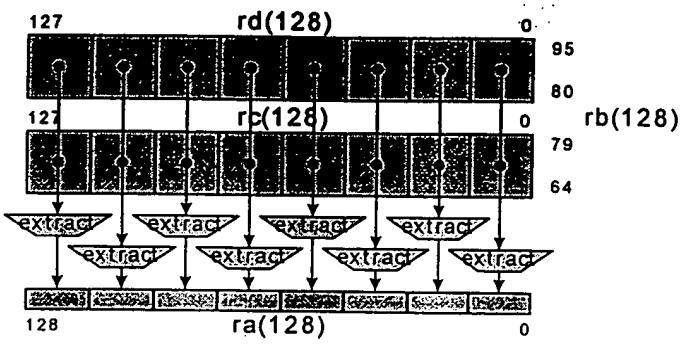


Ensemble complex multiply extract doublets

The ensemble-multiply-extract instructions (E.MUL.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 20D

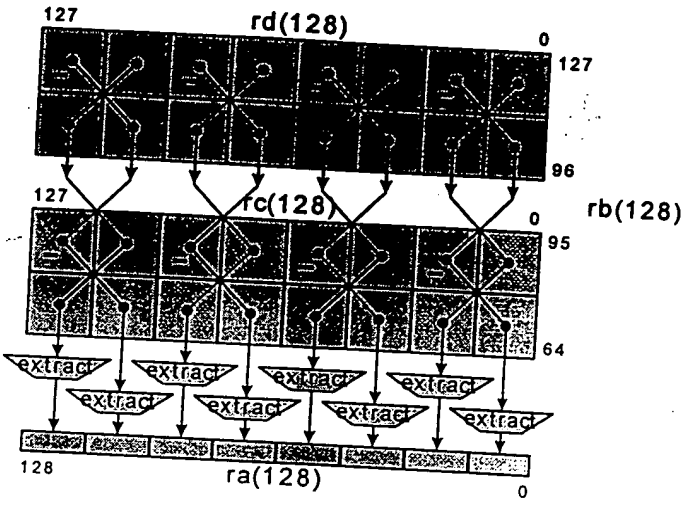
2040



Ensemble scale add extract doublets

FIG. 20E

2050

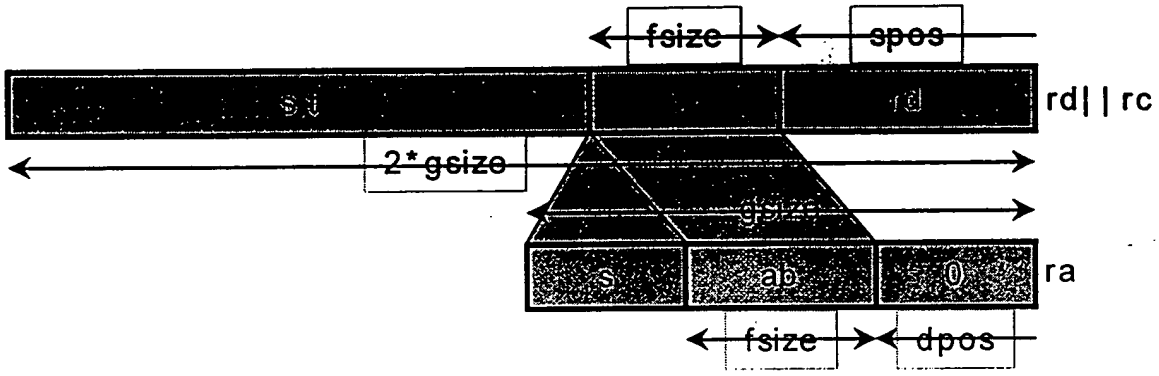


Ensemble complex scale add extract doublets

The ensemble-scale-add-extract instructions (E.SCLADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rd and rc registers by the rb register fields and produce extended (double-size) results.

FIG. 20F

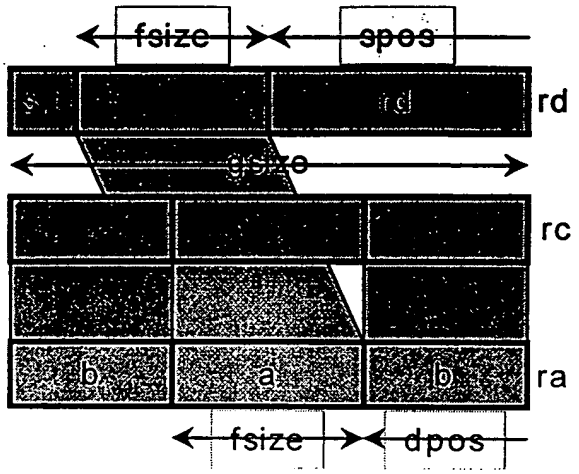
2060



Ensemble extract

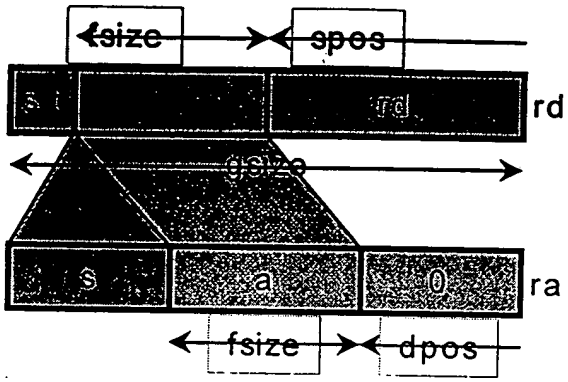
FIG. 20G

2070



Ensemble merge extract

FIG. 20H



Ensemble expand extract

FIG. 20I

Definition

```
def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&v_size-1+i)^h-size || v_size-1+i..i) * ((ws&w_size-1+j)^h-size || w_size-1+j..j)
enddef

def EnsembleExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b_8..0 of
        0..255:
            sgsize ← 128
        256..383:
            sgsize ← 64
        384..447:
            sgsize ← 32
        448..479:
            sgsize ← 16
        480..495:
            sgsize ← 8
        496..503:
            sgsize ← 4
        504..507:
            sgsize ← 2
        508..511:
            sgsize ← 1
    endcase
    l ← b_11
    m ← b_12
    n ← b_13
    signed ← b_14
    x ← b_15
    case op of
        E.EXTRACT:
            gsize ← sgsize*(2-(m or x))
            zsize ← sgsize
            h ← gsize
            as ← signed
            spos ← (b_8..0) and (gsize-1)
```

FIG 20J

```
        E.SCAL.ADD.X:
            if (sgsize < 8) then
                gsize ← 8
            elseif (sgsize*(n+1) > 32) then
                gsize ← 32/(n+1)
            else
                gsize ← sgsize
            endif
            ds ← cs ← signed
            bs ← signed ^ m
            as ← signed or m or n
            zsize ← gsize*(x+1)
            h ← (2*gsize) + 1 + n
            spos ← (b8..0) and (2*gsize-1)
        E.MUL.X:
            if (sgsize < 8) then
                gsize ← 8
            elseif (sgsize*(n+1)*(x+1) > 128) then
                gsize ← 128/(n+1)/(x+1)
            else
                gsize ← sgsize
            endif
            ds ← signed
            cs ← signed ^ m
            as ← signed or m or n
            zsize ← gsize*(x+1)
            h ← (2*gsize) + n
            spos ← (b8..0) and (2*gsize-1)
endcase
dpos ← (0 || b23..16) and (zsize-1)
r ← spos
sfsize ← (0 || b31..24) and (zsize-1)
tfsize ← (sfsize = 0) or ((sfsize+dpos) > zsize) ? zsize-dpos : sfsize
fsize ← (tfsize + spos > h) ? h - spos : tfsize
if (b10..9 = Z) and not as then
    rnd ← F
else
    rnd ← b10..9
endif
```

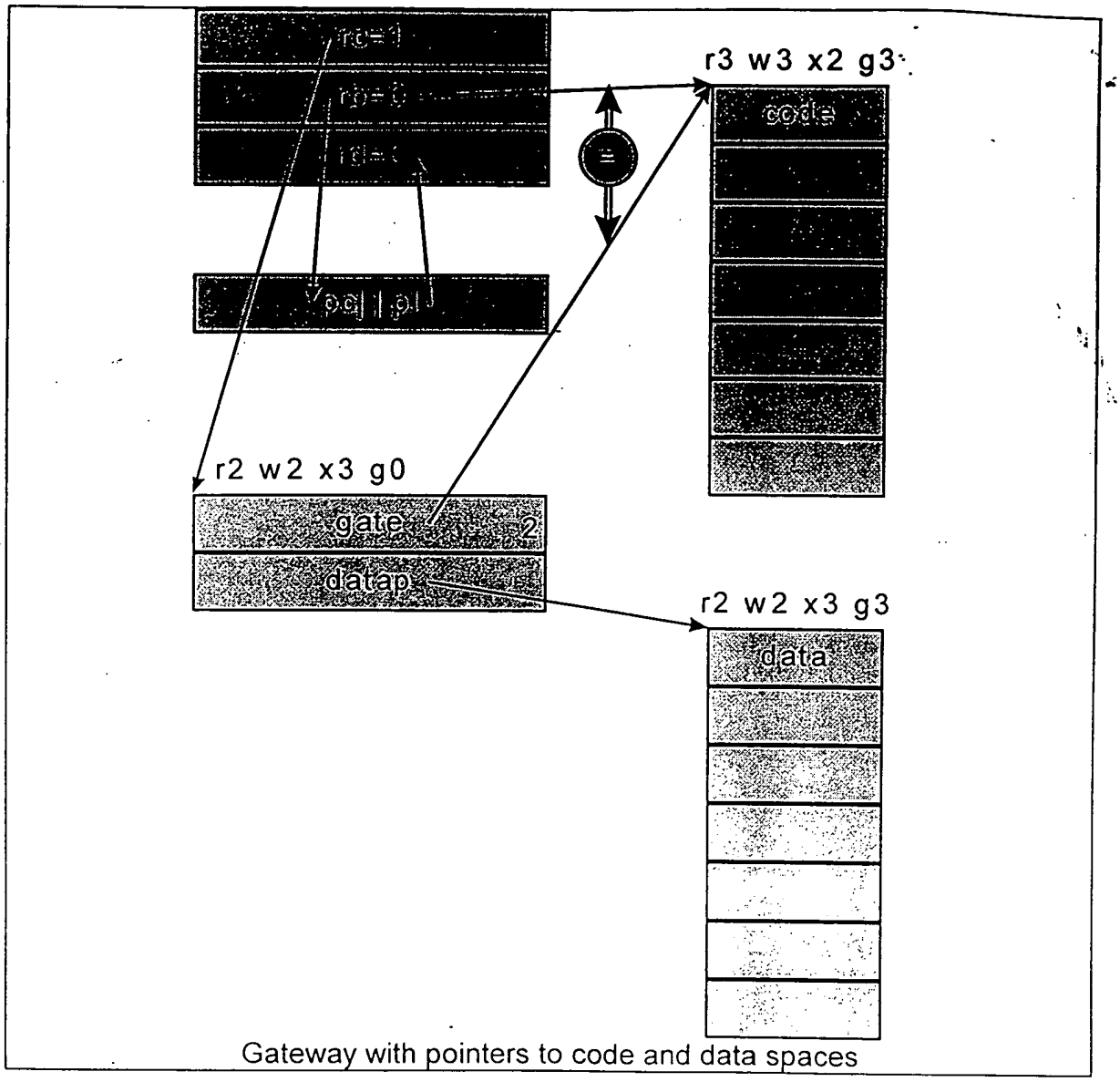
FIG. 20J (CONTINUED)

```
for j ← 0 to 128-zsize by zsize
    i ← j*gsize/zsize
    case op of
        E.EXTRACT:
            if m or x then
                p ← d_{gsize+i-1..i}
            else
                p ← (d || c)_{gsize+i-1..i}
            endif
        E.MUL.X:
            if n then
                if (i and gsize) = 0 then
                    p ← mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gsize,cs,c,i+gsize)
                else
                    p ←
mul(gsize,h,ds,d,i,cs,c,i+gsize)+mul(gsize,h,ds,d,i,cs,c,i+gsize)
                endif
            else
                p ← mul(gsize,h,ds,d,i,cs,c,i)
            endif
        E.SCAL.ADD.X:
            if n then
                if (i and gsize) = 0 then
                    p ← mul(gsize,h,ds,d,i,bs,b,64+2*gsize)
                        + mul(gsize,h,cs,c,i,bs,b,64)
                        - mul(gsize,h,ds,d,i+gsize,bs,b,64+3*gsize)
                        - mul(gsize,h,cs,c,i+gsize,bs,b,64+gsize)
                else
                    p ← mul(gsize,h,ds,d,i,bs,b,64+3*gsize)
                        + mul(gsize,h,cs,c,i,bs,b,64+gsize)
                        + mul(gsize,h,ds,d,i+gsize,bs,b,64+2*gsize)
                        + mul(gsize,h,cs,c,i+gsize,bs,b,64)
                endif
            else
                p ← mul(gsize,h,ds,d,i,bs,b,64+gsize) + mul(gsize,h,cs,c,i,bs,b,64)
            endif
    endcase
    case rnd of
        N:
            s ← 0^{h-r} || ~p_r || p_f^{r-1}
        Z:
            s ← 0^{h-r} || p_{h-1}^{r}
        F:
            s ← 0^h
        C:
            s ← 0^{h-r} || 1^r
    endcase
```

FIG. 20J (CONTINUED)

```
        v ← ((as & p_{h-1})||p) + (0||s)
        if (v_{h..r+fsize} = (as & v_{r+fsize-1})^{h+1-r-fsize}) or not (l and (op = E.EXTRACT)) then
            w ← (as & v_{r+fsize-1})^{zsize-fsize-dpos} || v_{fsize-1+r..r} || 0^{dpos}
        else
            w ← (s ? (v_h || ~v_h^{zsize-dpos-1}) : 1^{zsize-dpos}) || 0^{dpos}
        endif
        if m and (op = E.EXTRACT) then
            z_{zsize-1+j..j} ← c_{asize-1+j..dpos+fsize+j} || w_{dpos+fsize-1..dpos} || c_{dpos-1+j..j}
        else
            z_{zsize-1+j..j} ← w
        endif
    endfor
    RegWrite(ra, 128, z)
enddef
```

FIG. 20J (CONTINUED)



Gateway with pointers to code and data spaces

FIG. 21A

Typical dynamic-linked, inter-gateway calling sequence:

caller:

```
caller:  A.ADDI     sp@-size    // allocate caller stack frame
         S.I.64.A   lp,sp,off
         S.I.64.A   dp,sp,off
         ...
         L.I.64.A   lp=dp.off   // load lp
         L.I.64.A   dp=dp,off   // load dp
         B.GATE
         L.I.64.A   dp,sp,off
         ... (code using dp)
         L.I.64.A   lp=sp,off   // restore original lp register
         A.ADDI     sp=size     // deallocate caller stack frame
         B          lp          // return
```

callee (non-leaf):

```
calee:   L.I.64.A   dp=dp,off   // load dp with data pointer
         S.I.64.A   sp,dp,off
         L.I.64.A   sp=dp,off   // new stack pointer
         S.I.64.A   lp,sp,off
         S.I.64.A   dp,sp,off
         ... (using dp)
         L.I.64.A   dp,sp,off
         ... (code using dp)
         L.I.64.A   lp=sp,off   // restore original lp register
         L.I.64.A   sp=sp,off   // restore original sp register
         B.DOWN     lp
```

callee (leaf, no stack):

```
callee:  ... (using dp)
         B.DOWN     lp
```

FIG. 21B

Operation codes

B.GATE	Branch gateway

Equivalencies

B.GATE	← B.GATE 0

Format

B.GATE rb

bgate(rb)

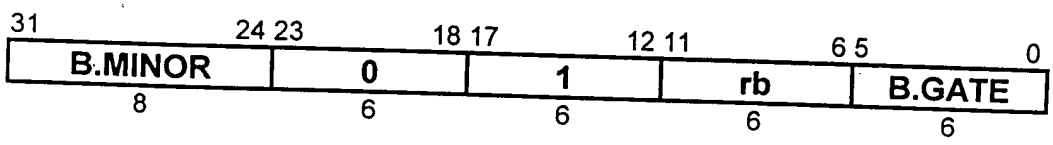
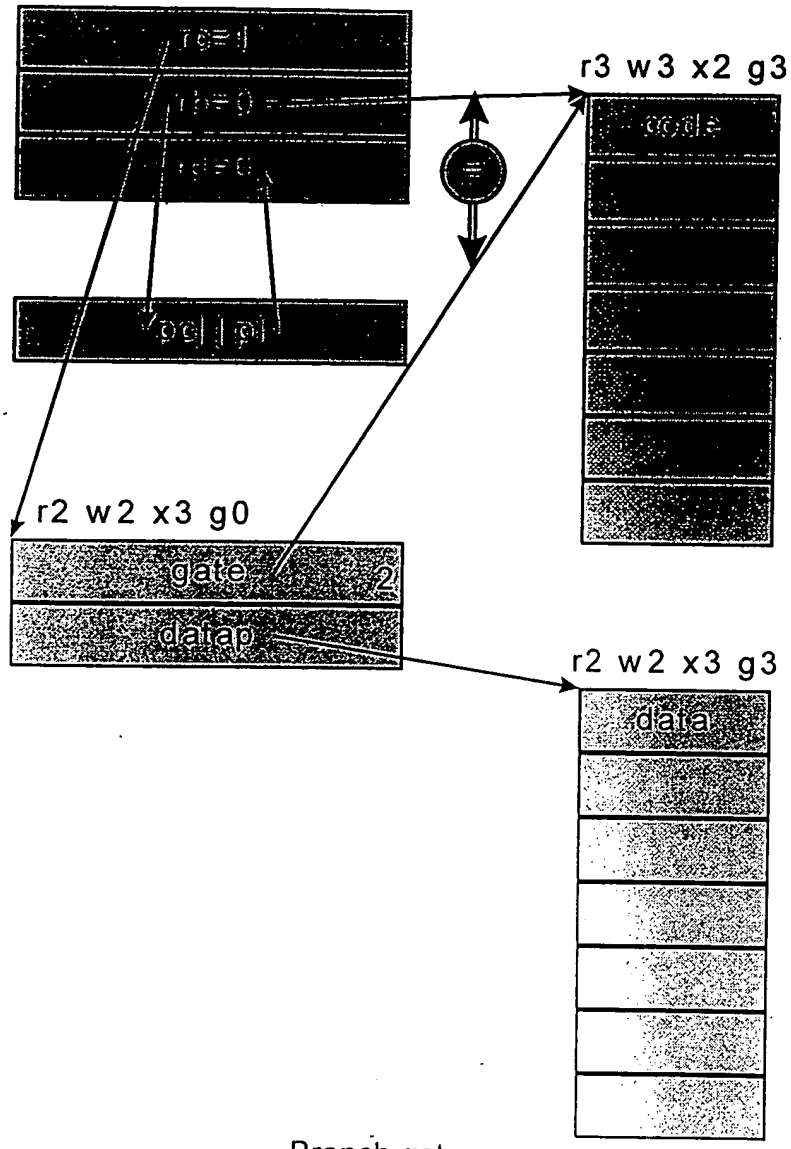


FIG. 21C



Branch gateway

FIG. 21D

Definition

```
def BranchGateway(rd,rc,rb) as
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 64)
    if (rd ≠ 0) or (rc ≠ 1) then
        raise ReservedInstruction
    endif
    if c2..0 ≠ 0 then
        raise AccessDisallowedByVirtualAddress
    endif
    d ← ProgramCounter63..2+1 || PrivilegeLevel
    if PrivilegeLevel < b1..0 then
        m ← LoadMemoryG(c,c,64,L)
        if b ≠ m then
            raise GatewayDisallowed
        endif
        PrivilegeLevel ← b1..0
    endif
    ProgramCounter ← b63..2 || 0^2
    RegWrite(rd, 64, d)
    raise TakenBranch
enddef
```

FIG. 21E

Exceptions

Reserved Instruction
Gateway disallowed
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 21F

Operation codes

E.SCAL.ADD.F.16	Ensemble scale add floating-point half
E.SCAL.ADD.F.32	Ensemble scale add floating-point single
E.SCAL.ADD.F.64	Ensemble scale add floating-point double

Selection

class	op	prec
scale add	E.SCAL.ADD.F	16 32 64

Format

E.op.prec ra=rd,rc,rb

ra=eopprec(rd,rc,rb)

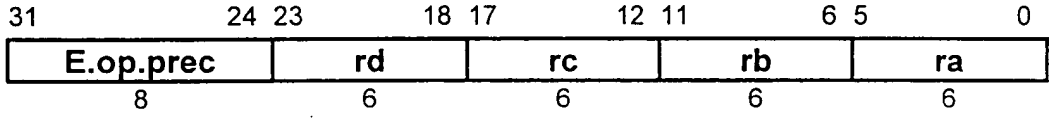


FIG. 22A

Definition

```
def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    for i ← 0 to 128-prec by prec
        di ← F(prec,di+prec-1..i)
        ci ← F(prec,ci+prec-1..i)
        ai ← fadd(fmul(di, F(prec,bprec-1..0)), fmul(ci, F(prec,b2*prec-1..prec)))
        ai+prec-1..i ← PackF(prec, ai, none)
    endfor
    RegWrite(ra, 128, a)
enddef
```

FIG. 22B

Operation codes

G.BOOLEAN	Group boolean

Selection

operation	function (binary)	function (decimal)
d	11110000	240
c	11001100	204
b	10101010	176
d&c&b	10000000	128
(d&c)|b	11101010	234
d|c|b	11111110	254
d?c:b	11001010	202
d^c^b	10010110	150
~d^c^b	01101001	105
0	00000000	0

Format

G.BOOLEAN rd@trc,trb,f

rd=gbooleani(rd,rc,rb,f)

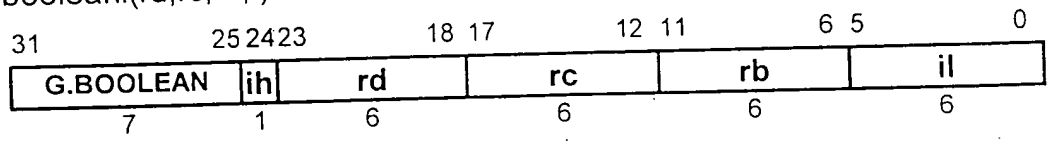


FIG. 23A

K 2320

```
if f6=f5 then
        if f2=f1 then
                if f2 then
                        rc ← max(trc,trb)
                        rb ← min(trc,trb)
                else
                        rc ← min(trc,trb)
                        rb ← max(trc,trb)
                endif
                ih ← 0
                il ← 0 || f6 || f7 || f4 || f3 || f0
        else
                if f2 then
                        rc ← trb
                        rb ← trc
                else
                        rc ← trc
                        rb ← trb
                endif
                ih ← 0
                il ← 1 || f6 || f7 || f4 || f3 || f0
        endif
else
        ih ← 1
        if f6 then
                rc ← trb
                rb ← trc
                il ← f1 || f2 || f7 || f4 || f3 || f0
        else
                rc ← trc
                rb ← trb
                il ← f2 || f1 || f7 || f4 || f3 || f0
        endif
endif
```

FIG. 23B

2330

Definition

```
def GroupBoolean (ih,rd,rc,rb,il)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    if ih=0 then
        if il5=0 then
            f ← il3 || il4 || il4 || il2 || il1 || (rc>rb)2 || il0
        else
            f ← il3 || il4 || il4 || il2 || il1 || 0 || 1 || il0
        endif
    else
        f ← il3 || 0 || 1 || il2 || il1 || il5 || il4 || il0
    endif
    for i ← 0 to 127 by size
        ai ← f(di||ci||bi)
    endfor
    RegWrite(rd, 128, a)
enddef
```

FIG 23C

2410

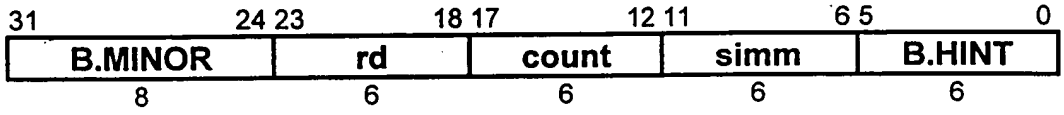
Operation codes

B.HINT	Branch Hint

Format

B.HINT badd,count,rd

bhint(badd,count,rd)



simm ← badd-pc-4

FIG. 24A

2430

Definition

```
def BranchHint(rd,count,simm) as
      d ← RegRead(rd, 64)
      if (d1..0) ≠ 0 then
            raise AccessDisallowedByVirtualAddress
      endif
      FetchHint(ProgramCounter +4 + (0 || simm || 02), d63..2 || 02, count)
enddef
```

FIG. 24B

246

Exceptions

Access disallowed by virtual address

FIG. 24C

Operation codes

E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16.C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default
E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
E.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default

FIG. 29A

2510

Selection

	op	prec				round/trap
integer from float	SINK.	16	32	64	128	NONE C F N X Z C.D F.D Z.D

Format

E.SINK.F.prec.rnd rd=rc

rd=esinkfprecrnd(rc)

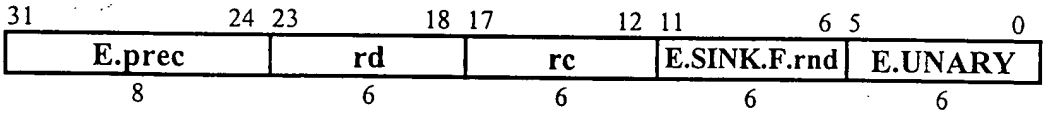


FIG. 2-A CONTINUED

2530

Definition

```
def EnsembleSinkFloatingPoint(prec,round,rd,rc) as
        c ← RegRead(rc, 128)
        for i ← 0 to 128-prec by prec
                ci ← F(prec,c_{i+prec-1..i})
                a_{i+prec-1..i} ← fsinkr(prec, ci, round)
        endfor
        RegWrite[rd, 128, a]
enddef
```

2560

Exceptions

Floating-point arithmetic

2570

```
Definition
def eb ← ebits(prec) as
        case pref of
                16:
                        eb ← 5
                32:
                        eb ← 8
                64:
                        eb ← 11
                128:
                        eb ← 15
        endcase
enddef

def eb ← ebias(prec) as
        eb ← 0 || 1^(ebits(prec)-1)
enddef

def fb ← fbits(prec) as
        fb ← prec – 1 – eb
enddef

def a ← F(prec, ai) as
        a.s ← ai_(prec-1)
        ae ← ai_(prec-2..fbits(prec))
        af ← ai_(fbits(prec)-1..0)
        if ae = 1^(ebits(prec)) then
                if af = 0 then
                        a.t ← INFINITY
                elseif af_(fbits(prec)-1) then
                        a.t ← SNaN
                        a.e ← -fbits(prec)
                        a.f ← 1 || af_(fbits(prec)-2..0)
                else
                        a.t ← QNaN
                        a.e ← -fbits(prec)
                        a.f ← af
                endif
        elseif ae = 0 then
                if af = 0 then
                        a.t ← ZERO
                else
                        a.t ← NORM
                        a.e ← 1-ebias(prec)-fbits(prec)
                        a.f ← 0 || af
                endif
        else
                a.t ← NORM
                a.e ← ae-ebias(prec)-fbits(prec)
                a.f ← 1 || af
        endif
enddef
```

FIG. 25D

```
def a ← DEFAULTQNAN as
        a.s ← 0
        a.t ← QNAN
        a.e ← -1
        a.f ← 1
enddef

def a ← DEFAULTSNAN as
        a.s ← 0
        a.t ← SNAN
        a.e ← -1
        a.f ← 1
enddef
```

```
def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
        if a.t=NORM and b.t=NORM then
                // d,e are a,b with exponent aligned and fraction adjusted
                if a.e > b.e then
                        d ← a
                        e.t ← b.t
                        e.s ← b.s
                        e.e ← a.e
                        e.f ← b.f || 0^(a.e-b.e)
                else if a.e < b.e then
                        d.t ← a.t
                        d.s ← a.s
                        d.e ← b.e
                        d.f ← a.f || 0^(b.e-a.e)
                        e ← b
                endif
                c.t ← d.t
                c.e ← d.e
                if d.s = e.s then
                        c.s ← d.s
                        c.f ← d.f + e.f
                elseif d.f > e.f then
                        c.s ← d.s
                        c.f ← d.f – e.f
                elseif d.f < e.f then
                        c.s ← e.s
                        c.f ← e.f – d.f
                else
                        c.s ← r=F
                        c.t ← ZERO
                endif
```

```
        // priority is given to b operand for NaN propagation
        elseif (b.t=SNAN) or (b.t=QNAN) then
                c ← b
        elseif (a.t=SNAN) or (a.t=QNAN) then
                c ← a
        elseif a.t=ZERO and b.t=ZERO then
                c.t ← ZERO
                c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
        // NULL values are like zero, but do not combine with ZERO to alter sign
        elseif a.t=ZERO or a.t=NULL then
                c ← b
        elseif b.t=ZERO or b.t=NULL then
                c ← a
        elseif a.t=INFINITY and b.t=INFINITY then
                if a.s ≠ b.s then
                        c ← DEFAULTSNAN // Invalid
                else
                        c ← a
                endif
        elseif a.t=INFINITY then
                c ← a
        elseif b.t=INFINITY then
                c ← b
        else
                assert FALSE // should have covered al the cases above
        endif
enddef

def b ← fneg(a) as
        b.s ← ~a.s
        b.t ← a.t
        b.e ← a.e
        b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef
```

```
def c ← fcom(a,b) as
    if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
        c ← U
    elseif a.t=INFINITY and b.t=INFINITY then
        if a.s ≠ b.s then
            c ← (a.s=0) ? G: L
        else
            c ← E
        endif
    elseif a.t=INFINITY then
        c ← (a.s=0) ? G: L
    elseif b.t=INFINITY then
        c ← (b.s=0) ? G: L
    elseif a.t=NORM and b.t=NORM then
        if a.s ≠ b.s then
            c ← (a.s=0) ? G: L
        else
            if a.e > b.e then
                af ← a.f
                bf ← b.f || 0^(a.e-b.e)
            else
                af ← a.f || 0^(b.e-a.e)
                bf ← b.f
            endif
            if af = bf then
                c ← E
            else
                c ← ((a.s=0) ^ (af > bf)) ? G : L
            endif
        endif
    elseif a.t=NORM then
        c ← (a.s=0) ? G: L
    elseif b.t=NORM then
        c ← (b.s=0) ? G: L
    elseif a.t=ZERO and b.t=ZERO then
        c ← E
    else
        assert FALSE // should have covered al the cases above
    endif
enddef
```

```
def c ← fmul(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e + b.e
        c.f ← a.f * b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO or b.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    else
        assert FALSE // should have covered al the cases above
    endif
enddef
```

```
def c ← fdivr(a,b) as
    if a.t=NORM and b.t=NORM then
        c.s ← a.s ^ b.s
        c.t ← NORM
        c.e ← a.e - b.e + 256
        c.f ← (a.f || 0^256) / b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← b.t
        c.e ← b.e
        c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
        c.s ← a.s ^ b.s
        c.t ← a.t
        c.e ← a.e
        c.f ← a.f
    elseif a.t=ZERO and b.t=ZERO then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=INFINITY then
        c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO then
        c.s ← a.s ^ b.s
        c.t ← ZERO
    elseif a.t=INFINITY then
        c.s ← a.s ^ b.s
        c.t ← INFINITY
    else
        assert FALSE // should have covered al the cases above
    endif
enddef

def msb ← findmsb(a) as
    MAXF ← 2^18 // Largest possible f value after matrix multiply
    for j ← 0 to MAXF
        if a_MAXF-1..j = (0^MAXF-1-j || 1) then
            msb ← j
        endif
    endfor
enddef
```

```
def ai ← PackF(prec,a,round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rn ← msb-1-fbits(prec) // lsb for normal
            rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
            rb ← (rn > rdn) ? rn : rdn
            if rb ≤ 0 then
                aifr ← a.f_{msb-1..0} || 0^{-rb}
                eadj ← 0
            else
                case round of
                    C:
                        s ← 0^{msb-rb} || (~a.s)^{rb}
                    F:
                        s ← 0^{msb-rb} || (a.s)^{rb}
                    N, NONE:
                        s ← 0^{msb-rb} || ~a.f_{rb} || a.f_{rb-1}
                    X:
                        if a.f_{rb-1..0} ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z:
                        s ← 0
                endcase
                v ← (0||a.f_{msb..0}) + (0||s)
                if v_{msb} = 1 then
                    aifr ← v_{msb-1..rb}
                    eadj ← 0
                else
                    aifr ← 0^{fbits(prec)}
                    eadj ← 1
                endif
            endif
            aien ← a.e + msb – 1 + eadj + ebias(prec)
            if aien ≤ 0 then
                if round = NONE then
                    ai ← a.s || 0^{ebits(prec)} || aifr
                else
                    raise FloatingPointArithmetic //Underflow
                endif
            elseif aien ≥ 1^{ebits(prec)} then
                if round = NONE then
                    //default: round-to-nearest overflow handling
                    ai ← a.s || 1^{ebits(prec)} || 0^{fbits(prec)}
                else
                    raise FloatingPointArithmetic //Overflow
                endif
            else
                ai ← a.s || aien_{ebits(prec)-1..0} || aifr
            endif
```

FIG. 2 (continued)

```
            SNAN:
                if round ≠ NONE then
                    raise FloatingPointArithmetic //Invalid
                endif
                if –a.e < fbits(prec) then
                    ai ← a.s || 1^ebits(prec) || a.f_a.e-1..0 || 0^fbits(prec)+a.e
                else
                    lsb ← a.f_a.e-1-fbits(prec)+1..0 ≠ 0
                    ai ← a.s || 1^ebits(prec) || a.f_a.e-1..-a.e-1-fbits(prec)+2 || lsb
                endif
            QNAN:
                if –a.e < fbits(prec) then
                    ai ← a.s || 1^ebits(prec) || a.f_a.e-1..0 || 0^fbits(prec)+a.e
                else
                    lsb ← a.f_a.e-1-fbits(prec)+1..0 ≠ 0
                    ai ← a.s || 1^ebits(prec) || a.f_a.e-1..-a.e-1-fbits(prec)+2 || lsb
                endif
            ZERO:
                ai ← a.s || 0^ebits(prec) || 0^fbits(prec)
            INFINITY:
                ai ← a.s || 1^ebits(prec) || 0^fbits(prec)
        endcase
    defdef
```

FIG. 20D

```
def ai ← fsinkr(prec, a, round) as
    case a.t of
        NORM:
            msb ← findmsb(a.f)
            rb ← -a.e
            if rb ≤ 0 then
                aifr ← a.f_{msb..0} || 0^{-rb}
                aims ← msb - rb
            else
                case round of
                    C, C.D:
                        s ← 0^{msb-rb} || (~ai.s)^{rb}
                    F, F.D:
                        s ← 0^{msb-rb} || (ai.s)^{rb}
                    N, NONE:
                        s ← 0^{msb-rb} || ~ai.f_{rb} || ai.f_{rb}^{rb-1}
                    X:
                        if ai.f_{rb-1..0} ≠ 0 then
                            raise FloatingPointArithmetic // Inexact
                        endif
                        s ← 0
                    Z, Z.D:
                        s ← 0
                endcase
                v ← (0||a.f_{msb..0}) + (0||s)
                if v_{msb} = 1 then
                    aims ← msb + 1 - rb
                else
                    aims ← msb - rb
                endif
                aifr ← v_{aims..rb}
            endif
            if aims > prec then
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (~as)^{prec-1}
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Overflow
                endcase
            elseif a.s = 0 then
                ai ← aifr
            else
                ai ← -aifr
            endif
        ZERO:
            ai ← 0^{prec}
        SNAN, QNAN:
            case round of
                C.D, F.D, NONE, Z.D:
                    ai ← 0^{prec}
                C, F, N, X, Z:
                    raise FloatingPointArithmetic // Invalid
```

```
                endcase
            INFINITY:
                case round of
                    C.D, F.D, NONE, Z.D:
                        ai ← a.s || (~as)^prec-1
                    C, F, N, X, Z:
                        raise FloatingPointArithmetic // Invalid
                endcase
        endcase
enddef
```

```
def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fsqr(fdiv(b,a)))
enddef

def c ← fest(a) as
    if (a.t=NORM) then
        msb ← findmsb(a.f)
        a.e ← a.e + msb - 13
        a.f ← a.f_msb..msb-12 || 1
    else
        c ← a
    endif
enddef

def c ← fsqr(a) as
    if (a.t=NORM) and (a.s=0) then
        c.s ← 0
        c.t ← NORM
        if (a.e_0 = 1) then
            c.e ← (a.e-127) / 2
            c.f ← sqr(a.f || 0^127)
        else
            c.e ← (a.e-128) / 2
            c.f ← sqr(a.f || 0^128)
        endif
    elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
        c ← a
    elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
        c ← DEFAULTSNAN // Invalid
    else
        assert FALSE // should have covered al the cases above
    endif
enddef
```

FIG 23D (CONTINUED)

Format

G.op.size rd=rc,rb

rd=gopsize(rc,rb)

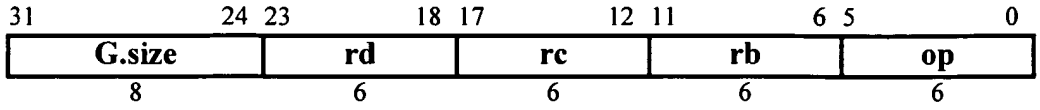


Fig. 26B

Operation codes

G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.ADD.128	Group add hexlet
G.ADD.L.8	Group add limit signed bytes
G.ADD.L.16	Group add limit signed doublets
G.ADD.L.32	Group add limit signed quadlets
G.ADD.L.64	Group add limit signed octlets
G.ADD.L.128	Group add limit signed hexlet
G.ADD.L.U.8	Group add limit unsigned bytes
G.ADD.L.U.16	Group add limit unsigned doublets
G.ADD.L.U.32	Group add limit unsigned quadlets
G.ADD.L.U.64	Group add limit unsigned octlets
G.ADD.L.U.128	Group add limit unsigned hexlet
G.ADD.8.O	Group add signed bytes check overflow
G.ADD.16.O	Group add signed doublets check overflow
G.ADD.32.O	Group add signed quadlets check overflow
G.ADD.64.O	Group add signed octlets check overflow
G.ADD.128.O	Group add signed hexlet check overflow
G.ADD.U.8.O	Group add unsigned bytes check overflow
G.ADD.U.16.O	Group add unsigned doublets check overflow
G.ADD.U.32.O	Group add unsigned quadlets check overflow
G.ADD.U.64.O	Group add unsigned octlets check overflow
G.ADD.U.128.O	Group add unsigned hexlet check overflow

Fig. 26A

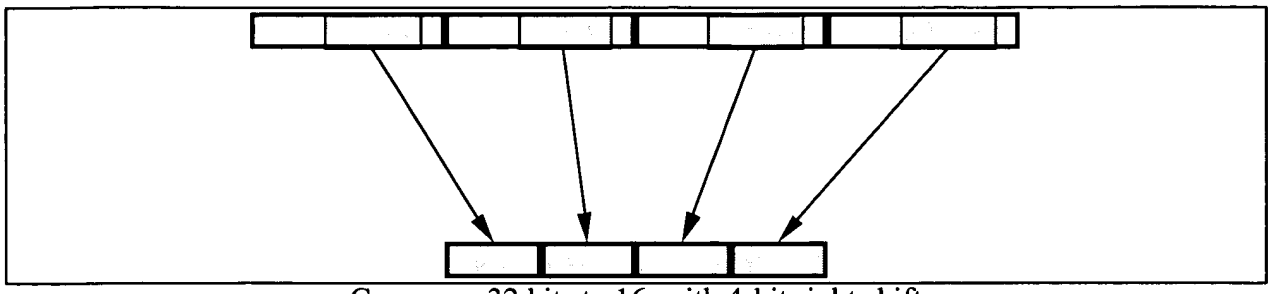
Definition

```
def Crossbar(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    shift ← b and (size-1)
    case op5..2 || 0^2 of
        X.COMPRESS:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a(i+hsize-1..i) ← c(i+i+shift+hsize-1..i+i+shift)
                else
                    a(i+hsize-1..i) ← c(i+i+size-1)^(shift-hsize) || c(i+i+size-1..i+i+shift)
                endif
            endfor
            a127..64 ← 0
        X.COMPRESS.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a(i+hsize-1..i) ← c(i+i+shift+hsize-1..i+i+shift)
                else
                    a(i+hsize-1..i) ← 0^(shift-hsize) || c(i+i+size-1..i+i+shift)
                endif
            endfor
            a127..64 ← 0
        X.EXPAND:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a(i+i+size-1..i+i) ← c(i+hsize-1)^(hsize-shift) || c(i+hsize-1..i) || 0^shift
                else
                    a(i+i+size-1..i+i) ← c(i+size-shift-1..i) || 0^shift
                endif
            endfor
        X.EXPAND.U:
            hsize ← size/2
            for i ← 0 to 64-hsize by hsize
                if shift ≤ hsize then
                    a(i+i+size-1..i+i) ← 0^(hsize-shift) || c(i+hsize-1..i) || 0^shift
                else
                    a(i+i+size-1..i+i) ← c(i+size-shift-1..i) || 0^shift
                endif
            endfor
        X.ROTL:
            for i ← 0 to 128-size by size
                a(i+size-1..i) ← c(i+size-1-shift..i) || c(i+size-1..i+size-1-shift)
            endfor
```

Fig. 32C

```
        X.ROTR:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← c_{i+shift-1..i} || c_{i+size-1..i+shift}
            endfor
        X.SHL:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← c_{i+size-1-shift..i} || 0^shift
            endfor
        X.SHL.O:
            for i ← 0 to 128-size by size
                if c_{i+size-1..i+size-1-shift} ≠ c^{shift+1}_{i+size-1-shift} then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← c_{i+size-1-shift..i}|| 0^shift
            endfor
        X.SHL.U.O:
            for i ← 0 to 128-size by size
                if c_{i+size-1..i+size-shift} ≠ 0^shift then
                    raise FixedPointArithmetic
                endif
                a_{i+size-1..i} ← c_{i+size-1-shift..i}|| 0^shift
            endfor
        X.SHR:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← c^{shift}_{i+size-1} || c_{i+size-1..i+shift}
            endfor
        X.SHR.U:
            for i ← 0 to 128-size by size
                a_{i+size-1..i} ← 0^shift || c_{i+size-1..i+shift}
            endfor
    endcase
    RegWrite(rd, 128, a)
enddef
```

Fig. 32C (cont'd)



Compress 32 bits to 16, with 4-bit right shift

Fig. 32D

Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

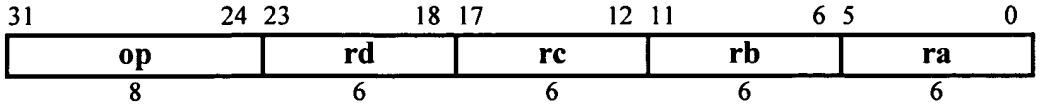
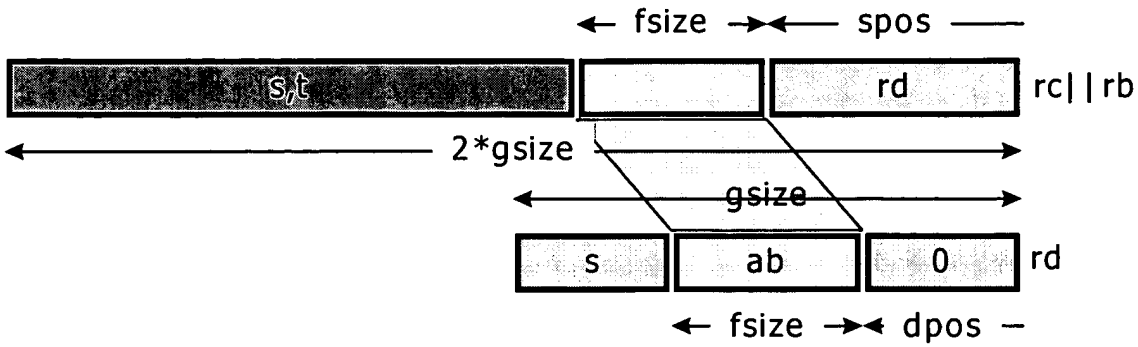


Fig. 33A

Fig. 33B

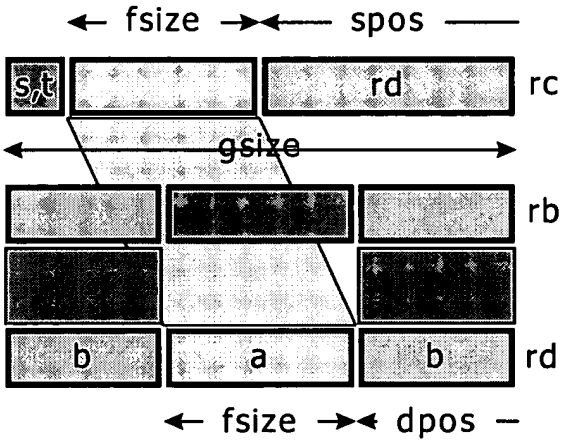
Definition

```
def CrossbarExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case b8..0 of
        0..255:
            gsize ← 128
        256..383:
            gsize ← 64
        384..447:
            gsize ← 32
        448..479:
            gsize ← 16
        480..495:
            gsize ← 8
        496..503:
            gsize ← 4
        504..507:
            gsize ← 2
        508..511:
            gsize ← 1
    endcase
    m ← b12
    as ← signed ← b14
    h ← (2-m)*gsize
    spos ← (b8..0) and ((2-m)*gsize-1)
    dpos ← (0 || b23..16) and (gsize-1)
    sfsize ← (0 || b31..24) and (gsize-1)
    tfsize ← (sfsize = 0) or ((sfsize+dpos) > gsize) ? gsize-dpos : sfsize
    fsize ← (tfsize + spos > h) ? h - spos : tfsize
    for i ← 0 to 128-gsize by gsize
        case op of
            X.EXTRACT:
                if m then
                    p ← dgsize+i-1..i
                else
                    p ← (d || c)2*(gsize+i)-1..2*i
                endif
        endcase
        v ← (as & ph-1)||p
        w ← (as & vspos+fsize-1)^gsize-fsize-dpos || vfsize-1+spos..spos || 0^dpos
        if m then
            asize-1+i..i ← cgsize-1+i..dpos+fsize+i || wdpos+fsize-1..dpos || cdpos-1+1..i
        else
            asize-1+i..i ← w
        endif
    endfor
    RegWrite(ra, 128, a)
enddef
```



Crossbar extract

Fig. 33C



Crossbar merge extract

Fig. 33D

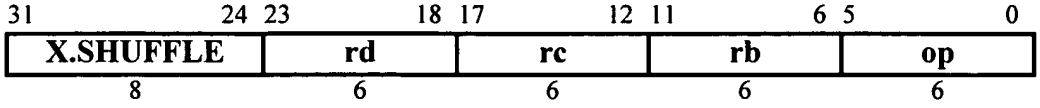
X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within triclet

Fig. 34A

Format

X.SHUFFLE.256 rd=rc,rb,v,w,h
X.SHUFFLE.size rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)
rd=xshufflesize(rcb,v,w)



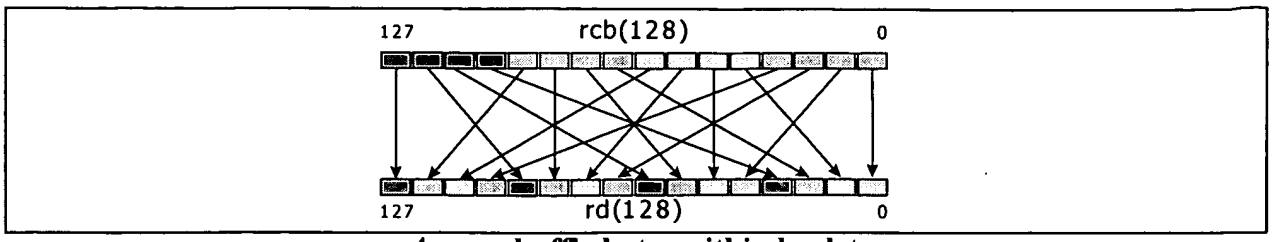
rc ← rb ← rcb
x←$\log_2$(size)
y←$\log_2$(v)
z←$\log_2$(w)
op ← ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) + (size=256)*(h*32-56)

Fig. 34B

Definition

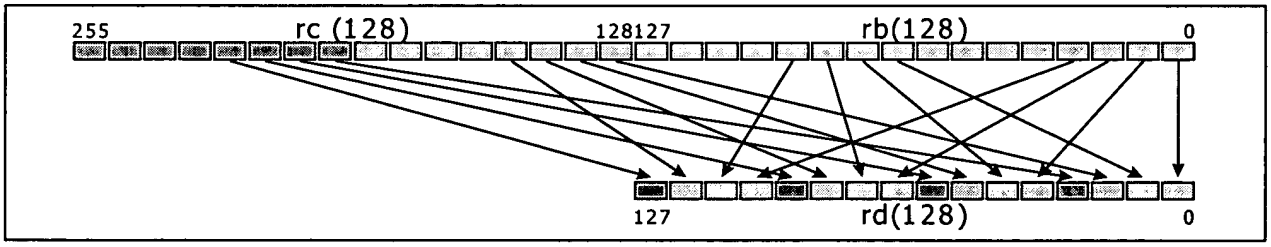
```
def CrossbarShuffle(major,rd,rc,rb,op)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    if rc=rb then
        case op of
            0..55:
                for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
                    if op = ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) then
                        for i ← 0 to 127
                            a_i ← c_(i_6..x || i_y+z-1..y || i_x-1..y+z || i_y-1..0)
                        end
                    endif
                endfor; endfor; endfor
            56..63:
                raise ReservedInstruction
        endcase
    elseif
        case op_4..0 of
            0..27:
                cb ← c || b
                x ← 8
                h ← op_5
                for y ← 0 to x-2; for z ← 1 to x-y-1
                    if op_4..0 = ((17*z-z*z)/2-8+y) then
                        for i ← h*128 to 127+h*128
                            a_i-h*128 ← cb_(i_y+z-1..y || ix-1..y+z || i_y-1..0)
                        end
                    endif
                endfor; endfor
            28..31:
                raise ReservedInstruction
        endcase
    endif
    RegWrite(rd, 128, a)
enddef
```

Fig. 34C



4-way shuffle bytes within hexlet

Fig. 34D



4-way shuffle bytes within triclet

Fig. 34E